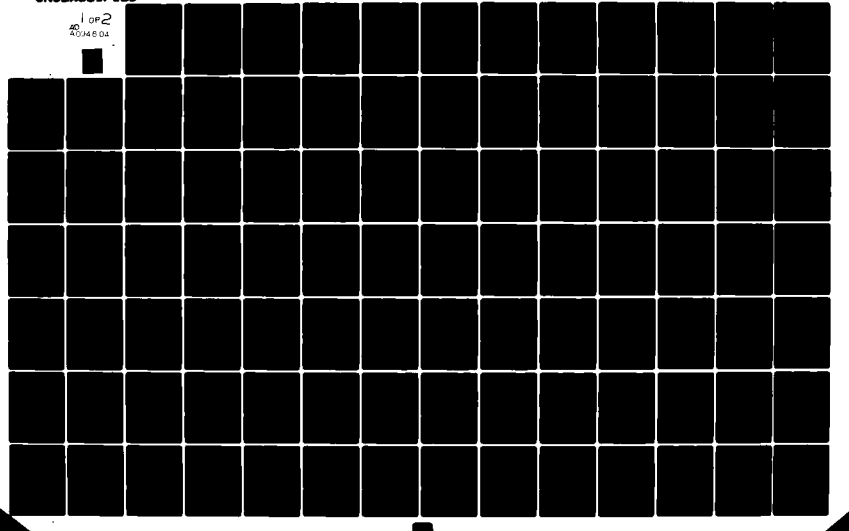
AD-A094 604 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
THEORY OF COMPILER SPECIFICATION AND VERIFICATION.(U)
MAY 80 W H POLAK MDA903-76-C-0206
UNCLASSIFIED NL



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: PVG-17	2. GOVT ACCESSION NO.: AD-A094604	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): THEORY OF COMPILER SPECIFICATION AND VERIFICATION,	5. TYPE OF REPORT & PERIOD COVERED: Thesis
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): W. H. Polak	8. CONTRACT OR GRANT NUMBER(s): MDA90376C0206 MCS7600321A1
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Stanford University Stanford, California	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: AO 2494
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: May 1980
	13. NUMBER OF PAGES: 193
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE. DISTRIBUTION UNLIMITED. THIS DOCUMENT MAY BE REPRODUCED FOR ANY PURPOSE OF THE U.S. GOVERNMENT

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

NA

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

COMPUTERS
COMPILER
PASCAL
PASCAL PLUS

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The formal specification, design, implementation, and verification of a compiler for a Pascal-like language is described. All components of the compilation process such as scanning, parsing, type checking, and code generation are considered.

The implemented language contains most control structures of Pascal, recursive procedures and functions, and jumps. It provides user defined data types including arrays, records, and pointers. A simple facility for input – output is provided.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The target language assumes a stack machine including a display mechanism to handle procedure and function calls.

The compiler itself is written in Pascal Plus, a dialect of Pascal accepted by the Stanford verifier. The Stanford verifier is used to give a complete formal machine checked verification of the compiler.

One of the main problem areas considered is the formal mathematical treatment of programming languages and compilers suitable as input for automated program verification systems.

Several technical and methodological problems of mechanically verifying large software systems are considered. Some new verification techniques are developed, notably methods to reason about pointers, fixed points, and quantification. These techniques are of general importance and are not limited to compiler verification.

The result of this research demonstrates that construction of large correct programs is possible with the existing verification technology. It indicates that verification will become a useful software engineering tool in the future. Several problem areas of current verification systems are pointed out and areas for future research are outlined.

Stanford Program Verification Group
Report No. PVG-17

Department of Computer Science
Report No. STAN-CS-80-802

May 1980

THEORY OF COMPILER SPECIFICATION AND VERIFICATION

by

Wolfgang Heinz Polak

Research sponsored by

Advanced Research Projects Agency
and
National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

THEORY OF COMPILER SPECIFICATION AND VERIFICATION

Wolfgang Heinz Polak

Department of Computer Science
Stanford University
Stanford, California 94305

May 1980

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, by the National Science Foundation under Contract MCS 76-00321-A1, and by the "Studienstiftung des deutschen Volkes."

Stanford Program Verification Group
Report No. PVG-17

May 1980

Department of Computer Science
Report No. STAN-CS-80-802

THEORY OF COMPILER SPECIFICATION AND VERIFICATION

by

Wolfgang Heinz Polak

ABSTRACT

The formal specification, design, implementation, and verification of a compiler for a Pascal-like language is described. All components of the compilation process such as scanning, parsing, type checking, and code generation are considered.

The implemented language contains most control structures of Pascal, recursive procedures and functions, and jumps. It provedes user defined data types includng arrays, records, and pointers. A simple facility for input-output is provided.

The target language assumes a stack machine including a display mechanism to handle procedure and function calls.

The compiler itself is written in Pascal Plus, a dialect of Pascal accepted by the Stanford verifier. The Stanford verifier is used to give a complete formal machine checked verification of the compiler.

One of the main problem areas considered is the formal mathematical treatment of programming languages and compilers suitable as input for automated program verification systems.

Several technical and methodological problems of mechanically verifying large software systems are considered. Some new verification techniques are developed, notably methods to reason about pointers, fixed points, and quantification. These techniques are of general importance and are not limited to compiler verification.

The result of this research demonstrates that construction of large correct programsis possible with the existing verification technology. It indicates that verification will become a useful software engineering tool in the future. Several

problem areas of current verification systems are pointed out and areas for future research are outlined.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206 and National Science Foundation under Contract NSF MCS 76-00321-A1. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

Abstract:

The formal specification, design, implementation, and verification of a compiler for a Pascal-like language is described. All components of the compilation process such as scanning, parsing, type checking, and code generation are considered.

The implemented language contains most control structures of Pascal, recursive procedures and functions, and jumps. It provides user defined data types including arrays, records, and pointers. A simple facility for input – output is provided.

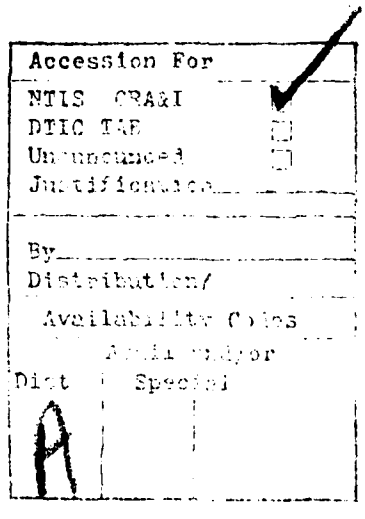
The target language assumes a stack machine including a display mechanism to handle procedure and function calls.

The compiler itself is written in Pascal Plus, a dialect of Pascal accepted by the Stanford verifier. The Stanford verifier is used to give a complete formal machine checked verification of the compiler.

One of the main problem areas considered is the formal mathematical treatment of programming languages and compilers suitable as input for automated program verification systems.

Several technical and methodological problems of mechanically verifying large software systems are considered. Some new verification techniques are developed, notably methods to reason about pointers, fixed points, and quantification. These techniques are of general importance and are not limited to compiler verification.

The result of this research demonstrates that construction of large correct programs is possible with the existing verification technology. It indicates that verification will become a useful software engineering tool in the future. Several problem areas of current verification systems are pointed out and areas for future research are outlined.



Preface

About three years ago David Luckham hinted to me the possibility of verifying a "real" compiler. At that time the idea seemed unrealistic, even absurd. After looking closer at the problem and getting more familiar with the possibilities of the Stanford verifier a verified compiler appeared not so impossible after all. In fact, I was fascinated by the prospect of creating a large, correct piece of software; so this subject became my thesis topic. I am very grateful to David Luckham for suggesting this topic and for his continued advice.

The research has drastically changed my view of verification and programming in general. Analysis and design of programs (even large ones) can be subject to rigorous mathematical treatment – the art of programming may become a science after all.

Naturally, the reader will be skeptical, still, I hope to be able to convey some of my fascination.

This work would have been impossible without the use of the Stanford verifier. I have to thank all members of the Stanford verification group for providing this excellent tool. Don Knuth's text processing system TEX was a most valuable asset for typesetting a manuscript that must be any typist's nightmare.

I thank my reading committee David Luckham, Zohar Manna, and Susan Owicki for their valuable time, for their careful reading, and their helpful advise. Friedrich von Henke contributed through numerous discussions and careful perusal of initial drafts of my writing. Last but not least I thank my wife Gudrun for her support.

Chapter I. Introduction

Chapter II. Theoretical framework

Chapter III. Source and target languages

Chapter IV. The compiler proof

Chapter V. Conclusions

Appendix 1. Formal definition of *LS*

Appendix 2. Formal definition of *LT*

Appendix 3. The Scanner

Appendix 4. The Parser

Appendix 5. Static semantics

Appendix 6. Code generation

Chapter I. Introduction

"The ultimate goals (somewhat utopian) include error-free compilers, ... ".

S. Greibach

1. Overview

In this thesis we describe the design, implementation, and verification of a compiler for a Pascal-like language. While previous work on compiler verification has focused largely on proofs of abstract "code generating algorithms" we are interested in a "real" compiler translating a string of characters into machine code efficiently. We consider all components of such a compiler including scanning, parsing, type checking and code generation.

Our interest is twofold. First, we are concerned with the formal mathematical treatment of programming languages and compilers and the development of formal definitions suitable as input for automated program verification systems. Second, we are interested in the more general problem of verifying large software systems.

There are several reasons for verifying a compiler. Compilers are among the most frequently used programs. Consequently, if we invest in program proofs it is reasonable to do this in an area where we may expect the highest payoff. Verification techniques are in general applied to programs written in high level languages. If we ever want to use verified programs we have to be able to correctly translate them into executable machine languages; another reason why correct compilers are a necessity.

The implemented language, *LS*, contains all features of Pascal [JW76] that are of interest to compiler constructors. The language contains most control structures of Pascal, recursive procedures and functions, and jumps. It provides user defined data types including arrays, records, and pointers. A simple facility for input - output is included; each program operates with one input and one output file.

The target language, *LT*, assumes a stack machine including a display mechanism [RR64, Or73] to handle procedure and function calls. This language simplifies our task somewhat as it avoids the issue of register allocation and similar irrelevant details. But at the same time the target language is realistic in that similar machine architectures exist, notably the B6700.

The compiler itself is written in Pascal Plus, a dialect of Pascal accepted by the Stanford verifier.[1] The Stanford verifier [SV79] is used to give a complete formal machine checked verification of the compiler.

We review existing methods for the formal definition of programming languages. We use the most appropriate definitional methods to formally define source and target language.

We further investigate how the correctness of a compiler can be specified in a form suitable for mechanical verification. Here, we have to deal with several technical issues such as fixed points and reasoning about pointers.

We show that an efficient program can be developed systematically from such specifications.

During this research verification has proven to be a most useful tool to support program development; verification should be an integral part of the development process and plays a role comparable to that of type ckecking in strongly typed languages. This methodology of verification supported programming is not limited to compilers, rather it is applicable to arbitrary problem domains.

The results of this thesis are encouraging and let us hope that verification will soon become a widely accepted software engineering tool. But also this work reveals many of the trouble spots still existing in today's verification technology. We point to several promising research areas that will make verification more accessible. The need for better human engineering and integrated software development systems is particularly urgent.

2. Program verification

Verification has provoked several controversial statements by opponents and proponents recently [DL79]. Therefore it is appropriate at this point to clarify what verification is, what it can do, and, most importantly, what it cannot do.

1.) Notable difference to standard Pascal is that formal documentation is an integral part of the language, for more details see 2.3.

To verify a program means to prove that the program is consistent with its specifications. Subsequently, the term "verification" is used as a technical term referring to the process of proving consistency with specifications. A program is "verified" if a consistency proof has been established.

Formal specifications for a program can express different requirements. For example, a specification can be "the program terminates for each set of input data". Or, even more trivially, one can specify that "the program satisfies all type and declaration constraints"; every compiler verifies this property. But of course, we consider more interesting properties of our compiler. What exactly its specifications are is discussed in the following sections and in more detail in chapter IV.

Since specifications can be weak and need not (and generally do not) express all requirements for a program verification should not be confused with correctness in the intuitive sense (i.e. the program does what one expects it to do). Verification is not a substitute for other software engineering techniques such as systematic program development, testing, walk-through and so on; rather verification augments these techniques. It gives us an additional level of confidence that our program has the property expressed in its specifications.

Depending on the application of a program certain errors may be mere annoyances while other may have disasterous effects. We can classify errors as "cheap" and "expensive". For example, in terms of our compiler cheap properties are the reliablility of error recovery and termination. An expensive error would be if the compiler would translate an input program without reporting an error but would generate wrong code.

As long as verification is expensive we can concentrate our efforts on the most costly requirements of a program. These requirements can be formalized and taken as specifications for the program. Verification can be used to guarantee these expensive requirements. Other cheaper properties can be validated by conventional testing methods. Redundant code can easily be added to a verified program to increase its reliability or establish additional properties without effecting verified parts of the program. For example, in our compiler a sophisticated error recovery could be added without invalidating program proofs.

2.1. Writing correct programs

The non-technical use of the expression "to verify a program" suggests that there is an existing program which we subject to a verification. This is possible in principle but most certainly not practical for large software systems;

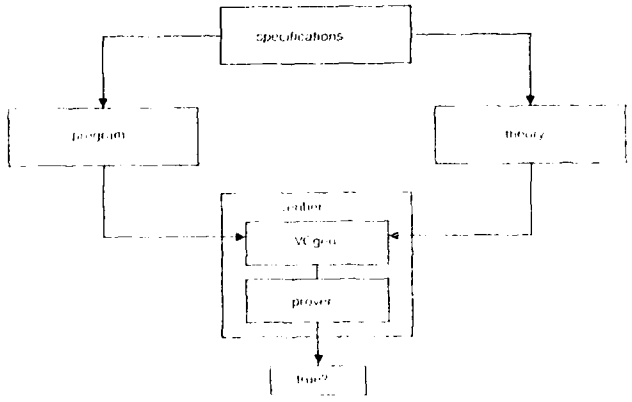


Fig. 1

it will only work for "toy" programs.

Instead, verification should be an integral part of program development: a program and its proof should be developed simultaneously from their specifications.

2.1.1. Program design

The process of designing a verified program can be visualized by figure 1. The starting point for the program design is a formal specification of the problem. In general, this specification alone is insufficient to prove the correctness of an implementation. We need additional theorems and lemmas about properties of the problem domain which enable an efficient implementation. We refer to these theorems and lemmas together with the formal specifications as "theory" or logical basis for our program proof.

The original specifications as well as derived lemmas and theorems are initially given in some suitable mathematical language. They have to be encoded in a form accepted by the verification system used. This step may not be as trivial as it appears on first sight. Problems arise if the "assertion language" used by the verification system is less expressive than the mathematical language. In the case of the Stanford verifier the assertion language is quantifier

free first order logic. The compiler on the other hand is specified using Scott's logic of computable functions. In chapter II we deal with the problem of encoding the language definition in our assertion language; of particular interest is the treatment of the least fixed point operator.

Given a suitable theory we implement a program in the style of structured programming [Di76]. Starting with a rough outline of the program we successively refine it until we derive an executable version. We start out with a first draft of our program where we are merely interested in the overall structure; all implementation details are omitted. The initial draft is then checked for consistency by the verification system. If consistency is not given, then either the program has to be corrected or stronger lemmas have to be added to the logical basis. After a successful verification we are sure that our initial design is correct (with respect to its specifications). Observe that this situation differs crucially from that in ordinary top down programming. In the latter case we have no means of validating an initial design until the refinement of the program is completed down to the lowest level and the program can actually be put on a machine and tested.

Given a verified first draft, we refine this program by implementing some of the points still left open. This step may require additional theorems and lemmas to be proven to enable efficient implementation and follows along the same lines as the initial design. We repeat this process until we arrive at an executable program.

2.1.2. An example

Let us explain the importance of the theory with a trivial *gcd* example. Our specification might be to implement a function $f(x,y)$ such that $f(x,y) = gcd(x,y)$. If we are given the following formal definition of *gcd* an efficient implementation of f is not easily derived:

$$gcd(x,y) = max\{z \mid x \bmod z = 0 \wedge y \bmod z = 0\}$$

The straightforward implementation suggested by this definition is to compute the set given in the definition and then search for the maximum. A much more efficient program is possible if we can prove additional theorems; for example $gcd(x,y) = gcd(x \bmod y, y)$.

In general, any efficient program is based on a mathematical property of the problem domain. The really creative part of programming is to uncover relevant properties of the specifications that lead to good implementations.

In many cases we can resort to well known results in literature. But note, that if we prove a theorem, then this is a purely mathematical activity and has

nothing to do with mechanical program verification. For example, in the case of the *gcd* above relevant theorems are part of number theory. In the future it is to hope that we have proof checking systems which can aid us in this part of program development.

People unfamiliar with verification often do not realize the importance of theory in writing programs. Programming is misleading in this respect because it allows us to write programs without having completely understood the problem. Verification makes us aware of all the mathematical properties (or alleged properties) we use in a program. A frequent complaint about verifications is that even simple problems require a discouraging amount of mathematics. We feel that this is inherent to programming: if we want correct programs we have to understand the problem and ultimately only a mathematical treatment can help us in this.

Also we should note that "toy" verification examples are often mathematically very subtle;[2] many programs written in practice are based on much simpler foundations.

2.1.3. A word of warning

A verified program may fail to execute safely for several reasons. Firstly, a program proof could be wrong. There are two main causes for this. It is most likely, that an error occured in manually proving theorems and lemmas that are the basis of the implementation. It can be hoped, that in the future these steps are amenable to automation or can at least be checked with proof checking systems. But then of course, there is the verification system itself which could be wrong. People have pointed out, that trying to verify the verifier is a vicious circle and cannot work. True, but, we have a much simpler way out. We can take our program with formal specifications and subject it to a different verification system. Unless both verifiers contain the same bug this will reveal any error.

Secondly, even if the program proof were valid the program will ultimately be executed in a real world environment with all its uncertainties.

Still, verification is an extremely promising tool to fight today's "software crisis". It is a tool to increase our confidence in a program to an extent not achievable with any other known means.

2.2. Logics of programming

So far we have discussed how to use verification during program develop-

2.) For example dealing with permutations [Po79]

ment. We now look into the details of automatic verification systems. First, we give a general overview over existing formal systems that allow mechanical verification and discuss their advantages and disadvantages. We then take a closer look at the Stanford verifier and describe its use.

The focus of our attention is on three methods for verifying programs. These methods are selected because they appear to be reasonably well developed and more importantly, there are implementations of verification systems based on these methods.

2.2.1. Logic of computable functions (LCF)

LCF [Mi72a] is a formal system based on Scott's logic of computable functions.[3] Since in LCF programs are objects of the formal language it is possible to express very general theorems about programs in LCF. Examples are program equivalence, program transformations as well as consistency and termination.

Although LCF is best suited to reason about denotational language definitions [AA74, AA77, He75] it is also capable of expressing Hoare style proof rules (see below).

Semi-automatic proof checking systems for LCF have been implemented [Mi72b, GM75] and used for several program proofs [MW72, Co79a, Co79b, Ne73].

2.2.2. First order logic

In [Ca76, CM79] McCarthy and Cartwright propose to consider recursive programs as objects in a first order language. The logical system has to be augmented with an undefined symbol to account for nonterminating computations; recursive function definitions are translated into equations in the logic. Suitable induction schemas allow reasoning about least fixed points. The system has the drawback that it is restricted to first order functions. Furthermore punning between logic and programming language severely restricts the languages that can be handled and at this point the method seems not suitable to prove properties of Algol-like programs. FOL [WT74] is a proof checking program that allows mechanized reasoning about first order logic. FOL has been used for several program proofs based on McCarthy's and Cartwright's method.

3.) Details of Scott's logic are discussed in chapter II.

2.2.3. Hoare's logic

In [Ho69] Hoare devised a deductive system[4] in which programs together with specifications can be derived. The verification problem is to determine if there exists a derivation of the given program together with its specifications. Given certain intermediate documentation (invariants) there is an effective way to construct a set of first order formulas which entail that such a derivation exists. Thus, if all these formulas (verification conditions) can be proven, the program is consistent with its specifications.

Hoare's logic is limited to partial correctness proofs; termination can be expressed by modifying the input program to include counters for all loops and suitable assertions as is outlined in [LS75, MP73]. Here, too, is a pun in that one considers expressions in the programming language to be terms in the underlying logic. This effectively excludes certain language constructs, e.g. expressions that cause side effects. The method requires the programmer not only to give input and output specifications but also specifications at intermediate points in the program.

2.2.4. Comparison

All of the methods mentioned above are interrelated. See for example [HL74, Do76, Po80].

Verification in this thesis is exclusively based on Hoare's logic and its implementation in the Stanford Verifier. This choice has been made because the method of expressing induction hypotheses as invariants in the program is natural and powerful. Furthermore the theorem proving part of the Stanford system is extremely sophisticated which is crucial in view of the size of the program we set out to verify.

The main disadvantage of using the Stanford verifier is that proofs of theorems from specifications (as outlined above) has to be done manually. In other systems, e.g. LCF and FOL the derivation of the necessary lemmas could be done within the system itself. Of course, we don't get this for free: in return we have to guide these system very heavily in order to find the necessary proofs. Supplying lemmas to the Stanford Verifier lets us easily resort to well known results in literature; but we should always keep in mind that this practice bears the possible danger of introducing errors in the proof by supplying erroneous lemmas.

4.) building on [Fl67]

2.3. The Stanford verifier

This is a brief overview over the Stanford verifier, subsequently referred to as "the verifier". For more details the reader should consult [SV79]. The verifier accepts standard Pascal with minor syntactic changes. In addition certain specifications are required, some are optional. The verifier uses the (formally) documented program to generate a set of first order formulas (verification conditions) based on a set of Hoare style proof rules [HW73]. If all verification conditions are true, the program is consistent with its specifications.

Note, that we distinguish "proving verification conditions" from "proving theorems and lemmas". The former is done completely mechanically by the verifier's prover while the latter refers to manual proofs. Occasionally we talk about "meta theorems". A meta theorem is a theorem about Hoare's logic itself. Examples are weak $\forall$-introduction and special proof rules about certain pointer operations to be introduced in later chapters.

In the language accepted by the verifier the following documentation exists:

- *entry* and *exit* assertions are required for each procedure and the main program describing input and output predicates.
- An *invariant* is required for each loop.
- All loops constructed with jumps have to be cut by at least one *assert* statement, however, additional *assert* statements may appear at arbitrary places in the program.
- A *comment* statement is similar to an *assert* statement except that it does not break the path through the program where it is attached, rather it requires additional specification to be proven (see [SV79]).

The assertion language is quantifier free first order logic with arbitrary function and predicate symbols. All free variables in assertions have to be variables of the program. Special terms are added to express operations on complex data structures.

- $< A, [i], e >$ where A is an array denotes A with $A[i]$ replaced by e.
- $< R, .f, e >$ where R is a record with field f denotes R with $R.f$ replaced by e.

A similar notation is used for operations on pointers [LS76]; this is discussed in chapter II in greater length.

An important concept is the use of "virtual" code. The use of "auxiliary" variables in program verification has been pointed out in earlier work [LS75].

The concept of virtual code is much more general however. A virtual variable is one whose value at no point influences the value of "real" variables. We put no restrictions on the use of virtual variables. They may appear at any point where real variables appear, subject only to the restriction that they don't influence the outcome of a real computation. Furthermore we allow for virtual procedures, functions, types etc. It is shown in chapter IV how virtual code is useful in expressing the correctness of an algorithm in a natural way.

The verifier's built-in theorem prover contains complete decision procedures for various theories and their combination [NO78]. In addition it allows the user to specify "rules" axiomatizing free function and predicate symbols. Each rule expresses some logical axiom. Through different syntactic formats of rules heuristic information is conveyed to the prover how to use this particular fact in a proof. The logical language used for rules is essentially the same as for assertions within the program except that free variables are considered universally quantified.

The set of axioms and lemmas expressed by the rules required for a program proof is called the *logical basis* of this proof. Proofs are relative to their logical basis.

There are three different rule formats accepted by the verifier's theorem prover. Each rule expresses a certain logical fact. In addition different rule formats imply certain heuristics to the theorem prover. We briefly describe, the logical contents of the different rule formats; we are not concerned with the heuristic aspect of rules.

A replace rule[5] of the form

$$\textbf{replace } T_1 \textbf{ where } F \textbf{ by } T_2$$

is equivalent with $F \Rightarrow T_1 = T_2$. The where clause may be missing in which case $F \equiv true$ is assumed. A rule of the form

$$\textbf{infer } F_1 \textbf{ from } F_2$$

means $F_2 \Rightarrow F_1$. Similarly, the rule

$$\textbf{from } F_1 \textbf{ infer } F_2$$

means $F_1 \Rightarrow F_2$. The last two rule formats may appear with a *whenever* clause. Whenever clauses contain heuristic information for the theorem prover; they do not change the meaning of a rule.

5.) The name replace rule is misleading since the theorem prover does not implement a "rewrite system".

3. Formal definition of programming languages

In order to be able to specify a compiler we need a method to formally define programming languages. Although there exists extensive literature on how to formalize language definitions this work seems not generally accepted; in most language designs informal english definitions prevail. The only exceptions are the by now generally accepted context free grammars. Many of the errors in current software can be attributed to a lack of specification. In the case of programming languages insufficient definitions leave ample room for major confusion of the user as well as the implementor. One result is the "try it out" attitude of many programmers who use the compiler rather than the manual as language definition. Also much of what is perceived as a compiler error is simply due to insufficient or ambiguous specification of languages.

In this section we briefly review various existing formal methods for the specification of programming languages. This section is far from being complete; rather the main emphasis is to motivate the particular choice of definitional formalisms used in the remainder of this thesis.

We adopt the common division of a language into syntax and semantics. The syntax describes how a program is externally represented as a string of characters while the semantics defines what a program means. Syntax is further subdivided into "micro syntax" or "token syntax" and the phrase structure of programs. Semantics is divided into "static" and "dynamic" semantics. Roughly, the former describes when a program is semantically valid; that is, it formalizes all type and declaration constraints. The dynamic semantics describes what the meaning of a semantically valid program is. Exactly what we mean by the meaning of a program depends very much on the particular formalism; we elaborate on this later. In addition to the components of a language definition outlined so far we introduce "tree transformations" which describe the relation between the phrase structure of a program and an "abstract" program which is input to the semantic analysis phase. This step may or may not be explicit in a particular formalism.

3.1. Syntax

3.1.1. Micro syntax

The external representation of languages is frequently described in terms of *identifiers*, *numbers*, *reserved words* and so on. The "micro syntax" of a language defines the syntax of these smallest syntactic entities, subsequently

called tokens.

A class of tokens (like identifiers) can in general be described as a regular language and in most textbooks a reference to this fact seems to settle this matter. But for a formal verification we have to be more specific and devise a precise definitional formalism.

Gries [Gr71]defines a scanner generator. The input to such a program can in some sense be considered to be the formal definition of the micro syntax. However, a more abstract theory is desirable and is developed in chapter III.

To define the scanner formally we define (a) how the input string is to be broken into substrings which represent tokens and (b) how to map each of these strings into a token. Furthermore we define how these two mappings are connected to define the micro syntax.

In the course of this research several ways of giving a comprehensive definition of a scanner have been investigated. It turns out that there is one main problem in most methods: the definition tends to be very large and is very likely to be less comprehensible than a program implementing the scanner. This of course makes the usefulness of the whole verification questionable. For example, the definition of the micro syntax in terms of a finite automaton by specifying a transition table is extremely incomprehensible and error prone.

We adopt the following definitional schema to describe the micro syntax. We define a finite number of regular languages L_i, each of which characterizes a set of tokens. For example the set of identifiers, the set of numbers, the set of delimiters etc. With each language we associate a semantic function S_i which maps elements of L_i into the corresponding token. The name "semantic function" may be confusing in this context; it is to be understood in the sense that the meaning of a character string is a token. For example, a string of digits denotes a numeral. S_i are defined recursively in the style of a denotational definition. Finally, the scanner is given as a function *scan* which is defined in terms of L_i and S_i. An intuitive operational description of *scan* is

- discard all characters which are not an initial substring of any L_i
- find the longest initial substring l of the input such that $l \in L_i$ for some i
- output the token $S_i(l)$ and repeat this process until the input is exhausted.

3.1.2. Phrase structure

Syntax is the best understood part of a language definition. In almost all cases the definition uses a context free grammar [AU72]. Various simplifying notations have been introduced, none of which alters the definitional power of

this formalism. The main deficiency of a grammar is that it merely defines the set of syntactically valid programs: it does not specify any particular output other than the derivation tree.

Several very efficient parsing methods are available. In particlular for this compiler we use *LR — Parsing*[Kn65]. The key idea is to have a parsing table and a program interpreting this table. In some sense this interpreter defines a semantics for parsing tables and the tables themselves could be considered the formal definition of the syntax.

In chapter II we define the notion of a labeled context free grammar. For any unambiguous labeled context free grammar we define a mapping from token sequences to parse trees; labels of productions become labels in these trees. In chapter III the syntax of *LS* is defined in terms of these concepts.

3.1.3. Tree Transformations.

To provide a mapping of derivation trees into more suitable format we introduce tree transformations, functions mapping trees into trees. With this mapping the desired output format for the parser can be specified conveniently

Tree transformations have been studied extensively in the literature [Kr74, Kr75, Ro71]. In particular DeRemer [De73, De74] studied their application in compiler construction. He views the whole translation process as a sequence of tree transformations. The starting point is a forest of singleton trees, the input characters and the result is the machine code. However, since transformations are purely syntactic they cannot serve as a formal definition of the semantics of the source language. A semantics is only given very indirectly by relating a program in source language to an (by definition) equivalent program in target language.

We define tree transformations using recursive functions on trees. We do not put any restrictions on these functions. It turns out, however, that for our source language the functions are of particularly simple nature and allow efficient implementation.

3.2. Semantics

Here the situation is completely different from that in the syntactic realm. Various different methods have been proposed, none of which appears to be generally accepted as a standard definition of semantics. We briefly outline several known methods for the definition of programming language semantics and motivate our particular choice of denotational semantics as a basis for compiler verification.

3.2.1. Operational semantics

The most straightforward way to specify what a program does is to define an interpreter executing the program. Methods following this line are classified as operational definitions [Mc62, Mc63, Re72]. A special case of operational semantics is the Vienna definition method [LW69, We72].

A principal problem with operational semantics is that it does not define the meaning of programs abstractly. Rather, the "meaning of a program can only be determined for a particular set of input data by executing the program with the interpreter. It is not possible to reason about termination of programs on the basis of an operational semantics: if a program loops on a given set of input data then so will the interpreter. But maybe the most serious disadvantage is that an interpreter precludes many implementations of the language. Implementations that do not execute the program in the same way the interpreter does are very hard to verify since it amounts to proving the strong equivalence of two programs, the compiler and the interpreter.

On the other hand operational semantics has several important advantages. It provides a "complete" definition and it is rather easy to comprehend.

3.2.2. Denotational semantics

In contrast mathematical or denotational semantics [Sc70, SS71, Te76, Go79] introduces the meaning of a program as an object, a mapping from input data to results. Thus it is more abstract and allows reasoning about programs in a much more general way, e.g. programs can be compared for equivalence.

We argue that denotational semantics is best suited as a basis for a compiler proof. Firstly, it is the most abstract way of defining a language. As an important consequence a denotational definition relates very easily with other definition methods; other methods are subsumed and equivalence can easily be shown. Secondly, Scott provided a theory for the sound description of higher order functions and self application. Finally denotational definitions have been successfully applied in the definition of several realistic programming languages; amoung these are Pascal [Te77a], Algol60 [Mo75a], Algol68 [Mi72c], SAL [MS76].

The semantics of our source language is defined analogously to that of Pascal in [Te77a]. In particular we use Tennent's separation of the semantics in static and dynamic semantics. However, there are some substantial differences in the way memory management is defined. Our definition is much more suitable as basis for a compiler proof.

3.2.3. Floyd - Hoare logic

In 1969 Hoare introduced an axiomatic system [Ho69] to reason about properties of programs. Since then various extensions to the method have been defined and theoretical results have been established. Of main interest are questions of soundness, completeness and the relation of an axiomatic definition to other methods [HL74, Do76, Co77, Cl79, GM79]. All these questions can only be answered with respect to a complementary definition of the language.

Although *axiomatics* is very well suited to reason about programs, only in exceptional cases can it serve as a formal definition of a language [Cl79, MG79]. In fact, the method gains much of its strength and usefulness because many subtle issues such as typechecking are purposely left out of consideration.

Even though it is conceivable to construct an axiomatic system which gives a complete detailed definition of all aspects of a language this seems not at all desirable. Attempts to extend axiomatics to completely define complex languages (of the order of Algol68) results in unnatural definitions [Sc78]; a semantics is only provided in a very indirect way.

It is unclear, how a compiler proof based on an axiomatic language definition can be given. A full second order logic seems necessary to prove statements of the form "*all input output predicates which are satisfied by the source* program are also satisfied by the target program".

3.2.4. Algebraic semantics

Algebraic semantics is based on the observation that the abstract syntax of a programming language is an initial algebra in a class of algebras C. Gouguen et al [GT77]define any other algebra in C to be a possible semantics of the language. The relation between (abstract) syntax and semantics is given by the unique homomorphism between the initial algebra of C and any other member of C. The importance of the algebraic approach is that it unifies most other approaches to semantics. The main problem, however, is the construction of suitable semantics algebras. Denotational semantics is one way of specifying such an algebra.

Most of our presentation could be phrased in algebraic terms. However, we consider this unnecessary for our purpose and prefer not to do so.

An algebraic approach is also useful in the verification of compilers. In this case Cohn's central theorem [Co65, BL70] states that in a suitable algebraic framework (speaking in terms of syntax trees) it is only to prove that terminal nodes are translated correctly to ensure the correctness of the complete translator. This idea has been previously used to prove compiling algorithms (see section on previous work).

For our compiler proof the algebraic approach is not well suited. It requires to talk about the concept of homomorphism which is extremely hard, if not impossible to formalize suitably for present automated verification systems. Instead we use *conventional computation induction* which in some sense is equivalent (though it is a less abstract notion).

Further details go beyond the scope of this thesis, the reader should consult [Co65, Go78, GT77, MW72, Mo72, Mo73, Sc76a, TW79, WM72].

3.2.5. Others

Some other methods for formalizing the semantics of programming languages have been proposed which cannot readily be classified in one of the above categories. Two relatively important ones are two-level or van Wijngaarden grammars [Wi69, Wi76]and attribute grammars [Kn68]both concepts are briefly described below.

The basic idea of vanWijngaarden grammars was to extend context free grammars to describe all context dependencies. However aspects of the dynamic semantics are still described informally in English. The main problems in using two-level grammars is that they are fairly difficult to relate to other formalisms. E.g the definition of mode equality in [Wi76], section 7.3.1 and an equivalent first order definition of mode equality are very hard to relate and an equivalence proof is extremely tedious. A further problem is, that for two-level grammars there is no straightforward way of constructing a parser. Most existing Algol68 implementations use a standard context free grammar for their parser and ad hoc procedures to enforce context dependencies.

Another formalism is attribute grammars, first introduced by Knuth in [Kn68] They have the distinct advantage that it is at least in principle possible to automatically generate an efficient compiler from a language definition in terms of attribute grammars. Several researchers have investigated this possibility and compiler generators are being constructed [Ka76a, Ka76b, Bo76, Cl179, De78]. The prime reason for not using this concept in this thesis is that it constitutes a description on a rather low level of abstraction. Also, an attribute grammar does not define a language; rather it relates it to another language (the machine code) by directly describing the translation process. The connection between a set of attributes and an intuitive understanding of a language is very remote. In recent research Bjørner [Bj77]attempts to automatically generate an attribute grammar from a denotational definition.

This approach may in the future lead to efficient automatically constructed compilers; at this time, however, it is not useful for the research presented here.

3.3. Machine languages

In principle the techniques used for the source language definition are applicable to the target language as well. However, for the purpose of writing a compiler we are not so very much interested in "parsing" a given program in machine language. Rather, we perform the opposite step. But this is trivial and we omit the definition of a particular external representation for the target language. Consequently, all we are interested in is the semantics of the target language. As the reader may expect, we use denotational semantics for this purpose.

3.4. Summary

Altogether we define the source language by specifying a function *smean* mapping character strings into their meaning. The meaning of a program is simply a mapping from (input) files to (output) files. If our language had a more complicated file structure a different notion of meaning would be required.

The function *smean* in turn is defined as composition of functions defining the micro syntax, syntax, tree transformations, static, and dynamic semantics respectively.

The micro syntax is defined in terms of a function *scan*, mapping a string of characters into a string of tokens. The syntax is given by the funtion *parse* mapping token strings into a syntax tree. The function *treetr* maps syntax trees into abstract syntax trees. The static semantics is a function *ssem* defined on abstract syntax trees. It is the identity function if the program in question is semantically valid, otherwise *ssem* returns *error*. Finally, *dsem* maps an abstract syntax tree into its meaning, i.e. a function from files to files. Thus the function *smean* is given by

$$smean = dsem \circ ssem \circ treetr \circ parse \circ scan.$$

The target language is defined by a function *tmean* mapping abstract code sequences into their meaning, again a mapping from input to output files.

The above functions are defined precisely in chapter III of this thesis. For now it suffices to assume their existence. We ignore the possibility of runtime and compile time errors for now.

4. Developing a verified compiler

Informally the correctness of our compiler can be specified as follows:

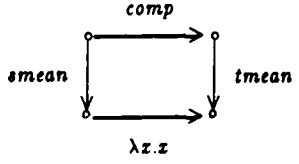
"Whenever the compiler terminates without printing an error message it has generated correct code"

In this section we show how this statement can be formalized, given a formal definition of source and target language.

The above statement is all we prove about the compiler; we do not prove that it ever terminates, that it prints reasonable error messages, etc. Still, the above property is extremely important; it guarantees that the user of the compiler is never deceived into believing his program is translated correctly if it is not.

4.1. What we prove

Source and target language are defined by the functions *smean* and *tmean* mapping a character string and a code sequence into their meaning. The compiler can be looked at as a function *comp* mapping a source program (a string of characters) into a target program (a code sequence). The compiler produces correct code if the following diagram commutes:



where $\lambda x.x$ is the identity function. In other words, the compiler is correct, if the meaning of the source program is equal to the meaning of the produced code. Although the above diagram is helpful in visualizing the correctness of a compiler it is not quite accurate in our case. The compiler to be written need not terminate for all input programs. Also, the compiler may issue meaningless error messages even if the input program is a valid *LS* program. If we consider printing of an error message as non-termination then the correct statement is "given that our compiler terminates then the above diagram commutes".

As described earlier, *smean* is given as a composition

$$smean = dsem \circ ssem \circ treetr \circ parse \circ scan.$$

According to this definition the compiler is divided into four (*treetr* and *parse* are computed in one program module) separate programs executed sequentially. Each program has an input domain and an output domain, such that output and input of two subsequent programs match.

The individual program modules can be thought of as procedures with the following specifications:

procedure *sc*(*in*: *char sequence*; **var** *out*: *token sequence*);
exit *out* = *scan*(*in*)

procedure *ps*(*in*: *token sequence*; **var** *out*: *abstract syntaxtree*);
exit *out* = *treetr*(*parse*(*in*))

procedure *ss*(*in*: *abstract syntactree*; **var** *out*: *abstractsyntaxtree*);
exit *out* = *ssem*(*in*)

procedure *cg*(*in*: *abstract syntaxtree*; **var** *out*: *code sequence*);
exit *tmean*(*out*) = *dsem*(*in*)

The compiler consists of the sequential execution of these four programs. This can be written as

$$sc(in, tks);\ pa(ts, as_1);\ ss(as_1, as_2);\ cg(as_2, out).$$

After executing this sequence it is true that

$$tsem(out) = dsem(ssem(treetr(parse(scan(in))))) \qquad (*)$$

By the standard interpretation for Hoare formulas this means that if the compiler terminates condition (*) holds. But (*) is just another way of stating that the diagram given above commutes.

A precise specification of the compiler and its components will be given in chapter III.

In the next subsection some principal techniques used in the verification of the compiler are discussed. Following this, we describe the concepts used in the implementation and proof of the individual components of the compiler.

4.2. Representation

Objects manipulated by a computer program are bits. If we are talking about integers, booleans, characters, or even sets or queues we actually are talking about a representation of these objects in terms of bits. For example, an integer number may be represented by a sequence of 32 bits; the value

of such a sequence is derived by interpreting the sequence as a binary representation. Alternatively, if we want to talk about arbitrarily large integers, we may have a linked list of elements of a base 1000 representation.

In general an abstract object is represented by specifying a data structure that is used to store such an object and by giving a mapping from this data structure to the desired object. To represent sets for example, we may choose linked list as representation and define a "representation function" as

$$setrep(l) = \textbf{if } null(l) \textbf{ then } \{\} \textbf{ else } \{car(l)\} \bigcup setrep(cdr(l)).$$

In chapter II we expand on this idea. In particular we are interested how function domains and functions on functions can be represented. In many cases the representation theory can be used to automatically select suitable data structures. A possible application are compiler generators, where data structures could be derived from domain definitions of the denotational semantics.

The idea of representation is relevant for all implementations [Jo79] and proofs. It significantly simplifies the proof of a refinement step, given a proof for the original program. The following trivial example explains this situation. Suppose we are given an abstract procedure *add* with the entry exit specification

$$\{A = A_0\}add(A, b)\{A = A_0 \bigcup \{b\}\}.$$

Given the simple theorem $(A \bigcup \{b\}) \bigcup \{c\} = A \bigcup \{b, c\}$ we can prove the correctness of the program

$$\{A = A_0\}add(A, b); add(A, c)\{A = A_0 \bigcup \{b, c\}\}.$$

Suppose now, that we implement sets as linked lists as above, then *add* operates on a list l instead of a set A. However, the documentation is still written in terms of sets. I.e. the refined version of *add* can be specified as

$$\{setrep(l) = A_0\}add(l, b)\{setrep(l) = A_0 \bigcup \{b\}\}.$$

Using the refined version of *add* does not change the structure of the proof of the using program. The only change is, that the documentation in the using program has to be changed to account for representation functions. The above example becomes

$$\{setrep(l) = A_0\}add(l, b); add(l, c)\{setrep(l) = A_0 \bigcup \{b, c\}\}.$$

No additional theorems are necessary to prove this program. Only in proving the implementation of *add* do we need properties of *setrep*.

4.3. Scanner

For our particular scanner we prove a theorem stating that for any two languages L_i and L_j their respective sets of initial substrings are disjoint. This is useful for an efficient implementation. For example seeing a letter in the input string means that this must be the beginning of an element of the set of identifiers and cannot belong to any other L_i.

In addition we need several trivial theorems about sequences of characters and tokens. They are of the nature of

$$\langle\rangle \cdot x = x \cdot \langle\rangle = x$$
$$x \cdot y = x \cdot z \rightarrow y = z$$

and deserve no further attention.

The implementation follows the recursive definition of *scan* and S_i and uses standard recursion removal techniques. In addition the above theorem about L_i allows several shortcuts to improve efficiency.

4.4. Parser

We use a LR parser [AJ74] to construct the required parse tree. We are not concerned with LR table construction; rather we assume that we have a correct table generator at our disposal.

The theory of LR parsing is fairly well understood; but a formal machine checked proof of an LR parser has not yet been published. The main problem we were faced with was to precisely define the properties of parsing tables required to prove the parser correct. More details are presented in chapter IV.

Again, of course, the parser heavily uses the representation theory and sequences as well as trees.

We talked so much about a stepwise development of program so the reader may be interested in some statistics. We have written and proven 10 successive versions of the parser. The starting point was a barebone version which assumed the correctness of parsing actions and did not produce output. The implementation of parsing actions followed. Next, in several versions we introduced the construction of parse trees and finally added the tree transformation step to produce abstract syntax trees.

The parser is a good example to illustrate the use of virtual code. It would of course be possible to have the parser construct a parse tree and after the program is parsed to apply tree transformations to produce the ultimate output. Instead, the parser produces a parse tree and an abstract syntax tree in parallel; part of the proof is to show that the abstract syntax is the image of

the parse tree under the tree transformations at any intermediate stage. We then can prove that the parser satisfies the following specifications

$$\{input = input_0\}pa\{pt = parse(input_0) \wedge ast = treetr(pt)\}.$$

It turns out, that the abstract syntax tree (*ast*) at no point depends on the parse tree (*pt*); rather it is derived directly from the parsing actions performed. Thus, the parse tree can be virtual code; it never has to be actually computed. The exit specification of the parser then implies that $ast = parse \circ treetr(input_0)$.

4.5. Semantic analysis

The static semantics is defined by recursive functions in Scott's logic of computable functions. In principle the implementation is straightforward but there are two theoretical problems.

The first problem is to express the definition in the assertion language accepted by the verifier. This is complicated by the fact that we have to deal with higher order functions (i.e. function mapping functions into functions) and that we have to find a consistent treatment of undefined elements($\perp$).

The second problem is what to do with the least fixed point operator (fix) that is used in the definition. If fix is used to define a recursive function then the solution is, of course, to compute this function recursively. The interesting case is, however, when fix is used to describe a data object which has to be computed.

The first problem can be solved fairly easily using the concept of representation functions; details are presented in chapter II.

The key idea to solve the second problem is referred to as "operationalisation" of fixed points. Let f be a function from D into D, we write $f \in D \rightarrow D$. The least fixed point of this functions $fix\ f$ is an element of D.[6] Given a representation for elements in D and a representation of $f \in D \rightarrow D$ the problem is to compute the representation of $fix\ f$. Note, if we are allowed to choose a suitable representation, this problem has always a solution. Just take the symbolic representation f to represent the function f, then $fix\ f$ is a symbolic representation of the fixed point. Of course, we are interested in a solution to this problem for more efficient representations.

Let us look at a simple example. Suppose the domain D is the set of all finite and infinite lists. As a representation we choose Lisp lists and define

$$rep(l) = \textbf{if } null(l) \textbf{ then } \langle\rangle \textbf{ else } \langle car(l)\rangle \cdot rep(cdr(l)).$$

6.) Precise definition of these notations can be found in chapter II.

Under *rep* a circular Lisp list represents an infinite abstract list. Clearly, not all infinite lists can be represented in this way; but a representation is not required to be surjective.

Let us now consider how we can compute the fixed point $fix(\lambda\ x.(a)\cdot x)$ which is a list consisting of infinitely many a's. First, we need a representation for functions from lists to lists. We take a pair of two pointers (S-expressions) for this purpose and define

$$funcrep(p,q) = \lambda\ l.\ \textbf{if}\ null(p)\ \textbf{then}\ \langle\rangle\ \textbf{else}\ \textbf{if}\ eq(p,q)\ \textbf{then}\ l\ \textbf{else}\ (car(p))\cdot funcrep(p,q)(l)$$

To represent $\lambda\ x.(a)\cdot x$ we simply take an arbitrary S-expression s other than *nil* and use the pair $(cons(a,s),s)$. Given an arbitrary pair (p,q), one can compute $\dot{p}$ such that $rep(\dot{p}) = fix(funcrep(p,q))$ as follows. Find a sublist t of p such that $eq(cdr(t),q)$; compute $rplacd(t,p)$; let $\dot{p} = p$. It is easy to prove that after these steps $\dot{p}$ has the desired property.

Applied to our example, we let $p = cons(a,s)$ and construct $rplacd(p,p)$. Clearly, p now points to a cyclic list which under *rep* represents an infinite list of a's.

In chapter II we formalize the above argument and prove a general theorem that then enables us to efficiently compute the least fixed points required for the semantic analysis.

Given the above theory, the implementation of the semantic analysis phase is straightforward.

4.6. Code generation

4.6.1. Necessary theory

We choose a fairly low level definition of the source language. For example, addressing is done relative to base addresses. For new procedure invocations new base addresses are set up. A display mechanism describes how memory of outer scopes is to be accessed. These concepts are familiar to the compiler writer and are well suited as our specification.

For some statements of *LS* we give a modified definition which is better suited for the verification of the code generation. For example, the definition of the while loop is originally given by a least fixed point. In the revised definition the meaning of the while loop is described by a conditional and explicit jumps. We prove manually that the revised definition is equivalent to the original one.

Also, we give a revised definition of the target language. We add the concept of blocks to the target language. All a block does is to hide labels defined inside from the outside. This concept is advantageous in generating code for example for the while loop. The necessary code requires introduction of labels in the target language that are not exsistent in the source language; it is convenient to be able to hide these in a block.

But we do not generate code for a different language. Rather, we prove that if all labels in a target program with blocks are distinct, then we can omit all blocks without changing the meaning of the program. The code generation produces "virtual" code containing blocks and real code that is identical except that blocks are omitted. The specification of the code generation is given as

$$\{ast = ast_0\}cg$$
$$\{tmean(code_1) = dsem(ast_0) \wedge code_2 = flat(code_1) \wedge distinct(code_1)\}\ .$$

$code_1$ contains blocks and is proven correct. $code_2$ is generated without blocks but is otherwise identical to $code_1$, expressed as $code_2 = flat(code_1)$. Finally, we prove that all labels in $code_1$ are distinct. Thus the above theorem is applicable and we may conclude $tmean(code_2) = dsem(ast_0)$. Observe, that this reasoning is anologous to that in the parser.

4.6.2. Implementation

With the above theory at hand the actual implementation is fairly simple. We have a set of procedures to generate fixed code sequences for primitive operations such as updating a location, accessing the value of a location and so on.

Code for more compound entities is generated recursively. Let us illustrate this with a very simple example. Suppose the recursive procedure *code* generates code for expressions. We have the specification

$$\{true\}code(E,z)\{\text{"}z \text{ computes } E\text{, puts result on stack"}\}.$$

Inside *code* we have a branch which deals with binary operations:

```
if E = "E1 + E2" then
        begin
            code(E1, z1);
            code(E2, z2);
            primcode(+, z3);
            z ← z1·z2·z3;
        end
```

primcode is a procedure generating code for primitive operations; in this case for "+" it produces code that takes the two top elements of the stack and pushes back their sum. Although this example is very much simplified, it demonstrates the basic principle of the code generation and its proof.

The reader may wonder about one more problem; how are fixed points of the semantic definition handled in the code generation. The solution is simple in this case. The meaning of a loop in the source language is described as a fixed point, say $fix\ s$. For this loop we generate code whose meaning in the target language is defined as a fixed point as well, say $fix\ t$. To show that both fixed points are equal we merely have to prove that $t = s$.

5. Related work

5.1. Previous work on compiler verification

Correct compilers have been of great interest to researchers from the very beginning of verification ideas. We summarize the work that has been done in this area and give a short characterization of the particular innovations introduced in each case.

The first attempt to verify a compiler or rather a translation algorithm, has been undertaken by McCarthy and Painter [MP66] in 1966 (see also [Pa67]). Though limited in scope this work has been of great impact and many researchers have directly built upon it. The language considered are arithmetic expressions. The compiler is a set of recursive functions and the proof proceeds by recursion induction.[7] The underlying semantical definition is operational.

A machine checked proof of the McCarthy/Painter compiler has been done by W. Diffie but has not been published.

Kaplan [Ka67] has built on McCarthy/Painter and proved a compiler for a yet more complex language including assignments and loops, still using the same proof technique. Again in [Bu69] R. Burstall proves the McCarthy/Painter compiler by structural induction. But as he points out, the proof is very similar to one given by recursion induction.

A completely new aspect enters the field of compiler verification in 1969 when Burstall and Landin [BL69] apply an algebraic method to greatly reduce the amount of work required in proving a compiler.

7.) different recursion principles are discussed in chapter II.

The same proof method is used by Milner and Weyhrauch [MW72, WM72] in 1972 to verify a compiler for an algolic language. But they introduce yet another innovation: semantics and proof are described in Scott's logic of computable functions. F.L. Morris uses the same algebraic method in [Mo72, Mo73]. More recently Thatcher, Wagner, and Wright [TW79] put forth similar ideas.

The proof techniques used by McCarthy and Painter are again employed by London [Lo71, Lo72] to prove an existing compiler for a simple version of LISP.

In recent years some authors tried to use axiomatic semantics as a basis for a compiler proof [Ch76, CM75]. But there appear to be some difficult problems. There is no notion of semantics in these proofs at all. Source and target languages are "defined" by a set of rules. In addition, static semantic issues are totally ignored. Again, the language handled is not much more interesting than that of the McCarthy/Painter compiler.

Lynn [Ly78] is the first to consider the problem of user defined functions in a compiler for LISP. He too uses Hoare style proof rules to specify the semantics of source and target language.

Yet a different approach is taken by Boyer and Moore. Their LISP theorem prover [BM77] has been shown to be capable of verifying an optimizing compiler for LISP expressions. Their system is capable of automatically synthesizing induction hypotheses for suitable recursive functions.

Two researchers continued to use denotational semantics as a basis for a compiler proof. First there are Milne and Strachey which in their book [MS76] give a proof of a compiling algorithm for a language of the complexity of Algol68. The proof is given completely by hand and the target language is a hypothetical machine language. Avra Cohn [Co79a, Co79b]proves the correctness of several components of a compiling algorithm using the Edinburgh implementation of LCF [GM75]. One problem she is interested in is the compilation of recursive procedures into stack implementations. Her main emphasis, though, is on the technique of using LCF and automating proofs in LCF.

A totally different approach is taken by H. Samet [Sa75]. Instead of proving the correctness of a compiler he proves equivalence of a particular source program and the produced code. Although this approach is in general much more expensive than verifying a compiler it is useful in situation where a correct translation is imperative and a correct compiler is not available.

5.2. Relation to our work

Most practitioners would not consider "compilers" verified previously to be "real compilers". Our emphasis is to verify a compiler that is - at least in principle - a practical, usable compiler. Therefore we choose a realistic source language in which a programmer might want to actually program. The only language comparable in size and complexity is *SAL*, considered by Milne and Strachey in [MS76]. But they consider a very abstract compiler only and their attention is limited to code generation, all proves are manual.

Milne and Strachey use a semantic definition of a higher level of abstraction than the definition of *LS* used in this work. If we were to use a more abstract definition a lower level definition would have to be developed as part of the theory for the code generation. The techniques and results in [MS76] are directly applicable for the necessary equivalence proof. In this sense the work in [MS76] is complementary to ours.

Cohn's work represents an important step towards mechanically checking proofs of the kind we give manually to derive theorems necessary for the code generation.

For our compiler to be practical we consider not only the code generation part but also verify a scanner, parser, and static semantic analysis. These issues have not been dealt with previously.

Although the development of a suitable logical basis for our compiler verification is done manually all actual program proofs are completely machine checked. No comparable mechanical verification attempt is known to the author.

An important part of this research deals with technical issues that are prerequisits for a mechanical compiler verification. These problems have not been considered before in this form. Some main points are

- formalization of compiler correctness suitable as input for a mechanical verfication system,
- structuring of programs such that mechanical verification becomes manageable, and
- development of specialized verification techniques to deal with pointer operations, least fixed points, and quantification.

5.3. Compiler generators

It would, of course, be much more efficient could we verify a compiler generator. We then could generate correct compilers for any specification.

But, given the current state of the art, this goal is somewhat unrealistic. We do not even know how to write compiler generators rather than how to verify them. Though much has been written about this subject no practical system exists to my knowledge. Most system called compiler generators turn out to be just parser generators.

Important work towards automatic generation of compilers has been done though We should mention P. Mosses' thesis [Mo75b] and the work based on attribute grammars [Ka76a, Ka76b, Bo76, CH79, Bj77].

Our work is not totally unrelated to compiler generation. Firstly, the parsing algorithm implemented operates table driven; thus it remains correct for any language that can be defined by an LR(1) grammar.

Further, we show how a program and its proof can be systematically derived from a denotational semantics. These results give some indication as to how this process or at least some parts of it can be automated.

6. Organization of this thesis

In chapter II we introduce notations and relevant theories required for the compiler proof. Among other things we consider

- first order logic and its application as assertion language
- Scott's logic of computable functions
- Axiomatization of Scottery in logic
- Proofs in logic and Scottery and their relation
- Theory of representation
- Treatment of fixed points

In chapter III we describe the source and target languages *LS* and *LT*. First, we give an informal introduction to the source language followed by the formal definition of scanner, parser, static and dynamic semantics. For each of these parts we define the definitional formalism used. We introduce a hypothetical machine executing the target language *LT*. A formal definition of *LT* is given.

Chaper IV describes precisely how the correctness of the compiler and its components is specified in a format accepted by the verifier. We then discuss the systematic development of the individual components and demonstrate the proof principles employed.

In the concluding chapter V we summarize our results and consider possible changes, improvements and extensions of the compiler. We discuss implications of this work on the design of verification systems and more sophisticated program development systems that encompass the whole design cycle of programs.

Chapter II. Theoretical framework

1. Basic concepts

1.1. Functions

Let A and B be two arbitrary sets; $A \mapsto B$ denotes the set of all total functions from A to B. "$\mapsto$" associates to the right. We assume that all functions are either constants or have exactly one argument. This allows to write function application in "curried" form as $f\ x$. This "juxtaposition" associates to the left; i.e. $f\ x\ y$ means $(f\ x)y$. Functions with several arguments as $f \in A \times B \mapsto C$ are applied to tuples as in $f(x,y)$. Alternatively, f can be defined for the isomorphic domain $A \mapsto (B \mapsto C)$ in which case we write $f\ x\ y$.

We use the semicolon (;) to break the normal precedence, i.e. $f; g\ h$ means $f(g\ h)$. Several terms separated by ; associate to the right, i.e. $T_1; T_2; T_3$ means $T_1(T_2(T_3))$.

New functions can be constructed from old ones by using composition, conditionals, and λ-abstraction as defined below.[1] This language is essentially Church's λ–calculus [Ch5¹]. However, we are not so much interested in the λ–claculus as a formal system; rather we use it as a notational vehicle.

[λ-abstraction] Let $T(x)$ be a term with the potentially free variable x, then $\lambda\ x.T(x)$ denotes a function f such that $f\ y = T(y)$.

The notations " let $x = T_1$ in T_2" and "T_2 where $x = T_1$" are both equivalent to $(\lambda x.T_2)T_1$.

Some special sets used are integer N and truthvalues $\{TT, FF\}$.

1.) We are not strictly formal here, rather we identify terms denoting functions with the function denoted. A formal treatment would require to introduce the semantics of the λ–calculus (e.g. [St77]).

[Conditionals] The conditional **if** T_1 **then** T_2 **else** T_3 has the usual meaning; it is only well defined if T_1 denotes an element in $\{TT, FF\}$. This definition of conditionals will be extended to domains in the next section.

[Redefinition] Let $f \in A \mapsto B$, then $f[x/y]$ denotes the function

$$\lambda\epsilon.\ \textbf{if } \epsilon = y \textbf{ then } x \textbf{ else } f\ \epsilon.$$

1.2. First order logic

We assume that the reader is familiar with basic concepts of logic (see e.g. Mendelson [Me64] or [Kl67]). The following defintion of first order logic is written in the style of denotational definitions and is mainly included here to give the reader an idea of the flavour of these definitions using a familiar example. Note, however, that all sets and functions are ordinary mathematical sets and total functions and do not have any added structure as they do in denotational semantics. This is possible since the domains in question are very simple and the problem of self application does not arise. Parentheses $[\![x]\!]$ are used to indicate that x is a syntactic object; this convention will be obeyed throughout this thesis.

1.2.1. Syntax

$x \in X$	infinite set of variables
$f_i \in Y_i$	i - ary function symbols
$p_i \in P_i$	i - ary predicate symbols
$t \in T = x \mid f_n(t_1, \ldots, t_n)$	Terms
$u \in F = true \mid false \mid t_1 = t_2 \mid p_n(t_1, \ldots, t_n) \mid u_1 \wedge u_2 \mid \neg u \mid \forall x.u$	Formulas

Terms $f_0()$ are abbreviated as f; if no ambiguities arise the arity index is omitted. Parentheses may be used arbitrarily to disambiguate. The formulas $u_1 \vee u_2$, $u_1 => u_2$, and $\exists x.u$ are abbreviations for $\neg(\neg u_1 \wedge \neg u_2)$, $\neg u_1 \vee u_2$, and $\neg(\forall x.\neg u)$ respectively. Some common function symbols like $+$, $-$, etc. are written infix.

1.2.2. Semantics

The semantics is defined by specifying the universe U of interpretations and "meaning" functions mapping terms into U and formulas into truth values. Formally we write:

U	Universe

$Fun_i = U \mapsto U \mapsto \ldots \mapsto U$	i - ary funtions
$Pre_i = U \mapsto \ldots \mapsto U \mapsto \{TT, FF\}$	i - ary predicates
$\phi \in \Phi = (X \mapsto U) \times (Y_i \mapsto Fun_i) \times (P_i \mapsto Pre_i)$	Interpretations

The last line, for example, is to be read as: "an interpretation ϕ is a function that maps variables into the universe U, i-ary function symbols into i-ary functions and i-ary predicate symbols into i-ary predicates.

$$\boxed{\mathcal{T} \in T \mapsto \Phi \mapsto U}$$

$$\begin{aligned}
\mathcal{T}[\![x]\!]\phi &= \phi[\![x]\!] \\
\mathcal{T}[\![f(t_1, \ldots, t_n)]\!]\phi &= (\phi[\![f]\!])(\mathcal{T}[\![t_1]\!]\phi)\ldots(\mathcal{T}[\![t_n]\!]\phi)
\end{aligned}$$

$$\boxed{\mathcal{F} \in F \mapsto \Phi \mapsto \{TT, FF\}}$$

$$\begin{aligned}
\mathcal{F}[\![true]\!]\phi &= TT \\
\mathcal{F}[\![false]\!]\phi &= FF \\
\mathcal{F}[\![t_1 = t_2]\!]\phi &= (\mathcal{T}[\![t_1]\!]\phi = \mathcal{T}[\![t_2]\!]\phi) \\
\mathcal{F}[\![p(t_1, \ldots, t_n)]\!]\phi &= (\phi[\![p]\!])(\mathcal{T}[\![t_1]\!]\phi)\ldots(\mathcal{T}[\![t_n]\!]\phi) \\
\mathcal{F}[\![u_1 \wedge u_2]\!]\phi &= \textbf{if } \mathcal{F}[\![u_1]\!]\phi \textbf{ then } \mathcal{F}[\![u_2]\!]\phi \textbf{ else } FF \\
\mathcal{F}[\![\neg u]\!]\phi &= \textbf{if } \mathcal{F}[\![u]\!]\phi \textbf{ then } FF \textbf{ else } TT \\
\mathcal{F}[\![\forall x.u]\!]\phi &= (\lambda\epsilon.\mathcal{F}[\![u]\!]\phi[\epsilon/x] = \lambda\epsilon.TT)
\end{aligned}$$

Other logical notions such as validity, satisfiability, proof, etc. can be defined in the obvious way in our formalism.

2. Scott's logic of computable functions

The mathematical theory underlying denotational semantics has been developed by Dana Scott [Sc70]. In lack of a better name we refer to this theory as Scott's logic of computable functions or Scottery for short. The theory we use differs slightly from that originally proposed by Scott; we assume an interpretation over complete partial orders (cpo's) [Pl78] rather than over lattices.

In this section we briefly summarize the relevant concepts and definitions of Scott's logic of computable functions. For a more complete treatment of this subject see [St77].

2.1. Basic definitions

[Quotients] Let S be a set and R be an equivalence relation on S, then the *quotient* S/R is the set of equivalence classes of R. If f is a function defined on S, then S/f is S/R where $R = \{(x, y) \mid f(x) = f(y)\}$.

[Directed set] If $(D, \sqsubseteq)$ is a partial order, then $S \subseteq D$ is *directed* if S has an upper bound.

[CPO] $(D, \sqsubseteq)$ is a *complete partial order* (*CPO*) if $(D, \sqsubseteq)$ is a partial order with

- bottom ($\bot_D \in D$) is the least element in D, and
- *every* directed subset S of D has a least upper bound ($\bigsqcup S$) in D.

[Domains] For the purpose of this thesis a domain is a CPO; we write D for a domain and omit the ordering $\sqsubseteq$ if it is clear from the context.[2]

Some domains have a set of *error elements* E; i.e. they are of the form $D + E$. We sometimes omit E in the definitions. Operations which are defined on D extend to $D + E$ in a strict way; that is they are the identity on E. If it is not necessary to distinguish different errors we use $E = \{?\}$.

[Flat Domains] Given an arbitrary set S the CPO $(S_\bot, \sqsubseteq)$ with $S_\bot = S \bigcup \{\bot\}$ and $a \sqsubseteq b$ iff $a = \bot$ is called a *Flat Domain*. Elements in S are called *proper*.

Flat domains are important because they allow us to construct a domain from a given set. Some typical examples we will use later are

$$T = \{TT, FF\}_\bot, \quad N = Z_\bot$$

where Z are the usual integers.

2.2. Operations on domains

[Sum] If D_1 and D_2 are two domains, then $D_1 + D_2$ is the *Sum* of D_1 and D_2. Elements of $D_1 + D_2$ are elements from $\{\langle 1, x\rangle \mid x \in D_1\} \bigcup \{\langle 2, x\rangle \mid x \in D_2\}$ with $\langle 1, \bot_{D_1}\rangle = \langle 2, \bot_{D_2}\rangle = \bot_{D_1+D_2}$. $\sqsubseteq$ extends to the sum according to $\langle i, x\rangle \sqsubseteq \langle j, y\rangle$ iff $i = j \wedge x \sqsubseteq y$.

If $D = D_1 + D_2$ and $x \in D_i$, then $x{:}D$ denotes $\langle i, x\rangle$, the injection of x into D. For $z = \langle i, y\rangle \in D$ the predicate $z \in D_j$ is true iff $i = j$. If $z = \langle i, y\rangle \in D$, then $z \mid D_j$ is the *projection* of z onto the component D_j, i.e.

$$\langle i, y\rangle \mid D_i = \begin{cases} \bot & \text{if } i \neq j \\ y & \text{if } i = j \end{cases}$$

Frequently the explicit projections and injections are omitted if no ambiguities arise.

[Product] $D_1 \times D_2$ is the *Product* domain of D_1 and D_2 with $(x_1, x_2) \sqsubseteq (y_1, y_2)$ iff $x_1 \sqsubseteq y_1 \wedge x_2 \sqsubseteq y_2$. If $x \in D_1 \times D_2$ then $x = (x^{\#1}, x^{\#2})$. Note, while x_i is a variable, "superscript #i" is a projection function defined on product domains.

Sum and product spaces extend to more than two dimensions in the obvious way.

[Continuous] A function $f \in D_1 \mapsto D_2$ is *continuous* if for all directed subsets $S \subseteq D_1$ we have $f(\bigsqcup S) = \bigsqcup f(S)$, where $f(S) = \{f(s) \mid s \in S\}$.

[Function domains] $D_1 \to D_2$ is the domain of continuous functions from D_1 into D_2 ordered by $f \sqsubseteq g$ iff $\forall x \in D_1 . f(x) \sqsubseteq g(x)$.[3]

[Monotonic] A function f is *monotonic* if $x \sqsubseteq y \to f(x) \sqsubseteq f(y)$.

Corollary: Every continuous function is monotonic. (Consider the directed set $S = \{x, y\}$ with $x \sqsubseteq y$.)

Corollary: If D_1 is a flat domain, then any monotonic $f \in D_1 \mapsto D_2$ is continuous. (All directed sets of a flat domain are trivial.)

[strict] A function $f \in D_1 \mapsto D_2$ is *strict* if $f(\bot_{D_1}) = \bot_{D_2}$.

Corollary: Any strict function defined on a flat domain is monotonic and continuous. (Immediate).

2.) Topological issues such as finite basis are irrelevant for our discussion. For more details see [Sc72a, Sc76b, Pl78].

3.) Note the difference between $A \mapsto B$ and $A \to B$.

[Strict product] Let R be the relation on $D_1 \times D_2$ such that

$$(x_1, x_2)R(y_1, y_2) \text{ iff } (x_1 = \bot_{D_1} \vee x_2 = \bot_{D_2}) \wedge (y_1 = \bot_{D_1} \vee y_2 = \bot_{D_2}).$$

The domain $D_1 \times_s D_2 = (D_1 \times D_2)/R$ is the *Strict Product* of D_1 and D_2.

Arbitrary expressions over domains using the operators $+$, $\times$, $\rightarrow$ define new domains. We use the convention that $\rightarrow$ associates to the right. $+$ and $\times$ are associative (up to isomorphism).

Equations defining new domains may be recursive (mutually). Domains defined recursively are well defined as is shown in [Sc76b].[4]

As an example consider lists L over a domain of atoms A. The domain L can be defined recursively as $L = A + A \times L$. Let us briefly investigate the structure of L. Clearly, L contains the chain

$$\bot \sqsubseteq \langle a, \bot\rangle \sqsubseteq \langle a, \langle a, \bot\rangle\rangle \sqsubseteq \langle a, \langle a, \langle a, \bot\rangle\rangle\rangle \sqsubseteq \ldots.$$

For L to be a cpo it has to contain the limit of this chain which is an infinite list. This is undesirable if, for example, we are only interested in finite lists as in LISP.

We get a different situation if we define $\dot{L} = A + A \times_s \dot{L}$, using the strict product. Now we have

$$\bot = \langle a, \bot\rangle = \langle a, \langle a, \bot\rangle\rangle = \langle a, \langle a, \langle a, \bot\rangle\rangle\rangle = \ldots,$$

thus at least this sequence does not generate infinite objects. In fact, it can be shown that $\dot{L}$ contains exactly the finite lists over A together with $\bot$; $\dot{L}$ is the flat cpo of finite lists over A.

Another example for domains with infinite objects are binary trees B with $B = B \times B + A$. Again, B contains infinite objects. Using strict products gives us exactly the finite binary trees.

[Fixed points] Let $f \in D \rightarrow D$, then $fix\ f$ denotes the least fixed point of f, that is the least element of $\{g \mid g = f(g)\}$.

For monotonic f the least fixed point exists; for continuous f the least fixed point is given by $fix\ f = \bigsqcup\{f_i\}$ where $f_0 = \bot$, $f_{i+1} = f(f_i)$.

2.3. Conditionals

Conditionals **if** — **then** — **else** appearing in formulas of Scott's logic

4.) The proof is based on a universal domain $U = U \rightarrow U$, other domains are defined as images of U under retracts.

are defined completely strict in the condition.

$$\textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2 = \begin{cases} \bot & \text{if } E_0 = \bot \\ E_1 & \text{if } E_0 = true \\ E_2 & \text{if } E_0 = false \end{cases}$$

The **if** construct may appear without an else part in which case it is defined as:

$$\textbf{if } E_0 \textbf{ then } E_1 = \begin{cases} \bot & \text{if } E_0 = \bot \\ E_1 & \text{if } E_0 = true \\ \bot & \text{if } E_0 = false \end{cases}$$

Both versions of the conditional are continuous.

2.4. Lists

Finite lists (or sequences) over a given domain occur so frequently that we introduce a special operator. If D is a domain then D^* is defined as $D^* = D + D \times_s D^*$.

We write $\langle\rangle$ for the empty sequence. $\langle x_1, \ldots, x_n\rangle$ is a sequence with elements $x_1, \ldots, x_n$. The function *hd* and *tl* yield the head and tail of a sequence and $\bot$ if applied to $\langle\rangle$.

A centered dot "$\cdot$" is used to denote concatenation. When no ambiguities arise we write $\langle x\rangle \cdot w$ as $x \cdot w$.

If s is a sequence $|\ s\ |$ denotes the length of s.

Let $f \in D_1 \rightarrow D_2$ be a function; unless otherwise stated $f^* \in D_1^* \rightarrow D_2^*$ is defined as

$$f^*\langle\rangle = \langle\rangle, \quad f^*(d_1 \cdot \ldots \cdot d_n) = f(d_1) \cdot f(d_2) \cdot \ldots \cdot f(d_n)$$

3. Induction proofs

The basic proof principle employed for Scott's computable functions is that of fixed point induction. In this section we describe fixed point induction and its relation to other induction principles, structural and recursion induction.

3.1. Fixed point induction

Fixed point induction [Pa69] appears in two disguises.[5] First, suppose we are given an object which is defined as the least fixed point of a function,

5.) Actually, both instances of fixed point induction are identical since recursively defined domains are images under recursively defined retracts [Sc76b].

say $f = fix\ h$. If we want to prove that a property P holds for f we can do this by proving $P(\perp)$ and $\forall x.P(x) => P(h\ x)$.

The second case obtains if we are given a recursively defined domain, say $D = \Psi(D)$, and wish to prove that property P holds for all elements in D. One can induct as follows:

- prove $P(\perp)$
- assuming $\forall x \in D.P(x)$ prove $\forall x \in \Psi(D).P(x)$.

In both cases it is necessary that property P is "admissible".

[admissible] P is *admissible* if and only if $P(\bigsqcup f_i)$ whenever $P(f_i)$ for all directed sets $\{f_i\}$.

To demonstrate the effect of admissibility consider $g = fix(\lambda\ x.a{\cdot}x)$, an infinite list of a's. Consider the non-admissible proposition $P \equiv \lambda\ x.x \neq a{\cdot}x$. Clearly, $P(\perp)$. Assuming $P(x)$, i.e. $x \neq a{\cdot}x$ it follows $a{\cdot}x \neq a{\cdot}a{\cdot}x$ thus we have $P(a{\cdot}x)$. But observe that P is not true for the fixed point g since by definition of the fixed point we have $g = a{\cdot}g$.

3.2. Recursion induction, structural induction

We now show, that both these induction principles are special cases of fixed point induction.

Let f be a recursively defined function. Recursion induction proves that a proposition P holds for all results of f, given that f terminates. One proceeds by proving P for non recursing calls of f (base case) and then proving P for arbitrary calls to f assuming that inner calls satisfy P.

The basic difference to fixed point induction is, that recursion induction proves P only for terminating calls to f. Fixed point induction is applicable to nonterminating programs as well. In Scottery the result of an infinite computation is not necessarily undefined; rather is the limit point of an infinite sequence of partially computed values [Sc72b]. Since limit points in form of nonterminating computations are not considered in recursion induction, admissibility is not required.

Structural induction corresponds to fixed point induction on data structures. Again, the difference to fixed point induction is, that structural induction does not deal with infinite objects. It assumes that the domain in question is "well founded" [Ma74]. A domain is well founded if it has no infinite descending chains. For example, the domain of all finite lists is well founded, while the domain of all finite and infinite lists is not. In general, $D = \Psi(D)$ is well founded if Ψ is strict. Since no infinite objects are present in a well founded set, admissibility is not required for structural induction.

Our definition of source and target languages contains both domains that admit structural induction and those that don't. The domains of the abstract syntax as well as modes allow structural induction, i.e. do not require admissibility. The domain of types[6] contains infinite chains in form of recursively defined types and structural induction is not applicable.

4. Verification techniques

We assume the reader to be familiar with the principles of Hoare's logic [Ho69] as well as the use of automated verification systems, such as the Stanford verifier [IL75, SV79].

This section deals with three problem areas which are not too well understood yet. We introduce new techniques to deal with pointers, quantification, and axiomatization of Scottery.

In general reasoning about pointers is extremely tedious. To isolate special simple cases we classify pointer operations as "reading", "updating", and "extension" operations; we prove a theorem that greatly simplifies reasoning about extension operations.

The second problem is that of expressing quantified formulas in a quantifier free assertion language. We present a deduction step iin Hoare's logic called "weak $\forall$-introduction", comparable to $\forall$-introduction in first order logic.

Thirdly, we discuss the axiomatization of Scott's computable functions. This requires a satisfactory treatment of the undefined element ($\perp$) as well as higher order functions.

4.1. Pointers

4.1.1. Pointers in the Stanford verifier

A satisfactory formal treatment of pointers has first been proposed by Luckham and Suzuki in [LS76]. The basic idea is to associate a "reference class" with each pointer type. The reference class is the collection of data objects pointed to by pointers of this type. The meaning of data structures involving pointers is explained relative to a reference class. The operation of dereferencing is described as "indexing" the reference class with a pointer. In

6.) abstract syntax, types, and modes are discussed in chapter III.

fact, the relation of pointers and reference classes is very much like that of indices and arrays.

Formally the expression $p\uparrow$ means $r \subset p \supset$ for the reference class r associated with the type of p. Assignments to data objects pointed to by pointers, e.g. $p\uparrow \leftarrow e$, are described as assignments to a whole reference class, e.g. $r \leftarrow < r, \subset p \supset, e >$. Similar to the array term $< a, [i], e >$ the term $< r, \subset p \supset, e >$ denotes the reference class r after $r \subset p \supset$ has been replaced by e. Assignment and selection obey the well known axioms

$$< r, \subset p \supset, e > \subset q \supset = \textbf{if } p = q \textbf{ then } e \textbf{ else } r \subset q \supset$$
$$< r, \subset p \supset, r \subset p \supset > = r$$

Reference classes are not part of Pascal; the Stanford verifier introduces a *reference class* $\#t$ automatically whenever the program contains a type declaration of the form $pt = \uparrow t$. In later language designs reference classes have been made explicit, e.g. Euclid [Lo78].

In addition to assignment and selection the "extension" operation $r \bigcup p$ is defined for reference classes. This construct is used to describe the effect of generating a *new* data object; it extends an existing reference class r by the new cell $p\uparrow$ pointed to by p.

4.1.2. Reasoning about pointers

Suppose we are given a reference class r and a pointer p and we know that p points to the beginning of a linked list. If we now add a new element to this list for example, then this will change the reference class r. A central problem in reasoning about reference classes is to prove that this change does not destroy the already existing list structure. Proof techniques for this case have been outlined in [LS75]. However, these techniques are fairly tedious and the resulting complexity is unacceptable for our compiler proof.

We need more abstract operations on pointers that are easier to deal with. An obvious idea is to use the concept of data abstraction [Gu75, GH76] to encapsulate pointer operations and define modules which provide abstract list or tree operations. Still, the implementation of such a module has to proceed on a very low level and poses all of the above problems.

Another approach has been proposed recently [Su80]. Suzuki introduces the idea of pointer rotations. However, this method is not complete in the sense that all pointer operations could be expressed as rotation. Also, proof techniques for dealing with pointer rotations have yet to be developed.

Our approach to pointers is as follows. We distinguish three different operations on pointer structures, "reading", "updating", and "extending".

Special proof techniques for these three cases are described below.

"Reading" data stored in a pointer structure (e.g. linked list) causes no problem whatsoever; no change of reference classes occurs and thus all structures are preserved.

"Updating" on the other hand is extremely difficult to handle. An update operation is one which changes an existing reference class. The situation is familiar to the lisp programmer using language features like RPLACA and RPLACD. Update operations are only used at one place throughout the compiler proof. For this special case we prove that the particular update operation performed corresponds to the computation of a least fixed points. This situation is discussed in detail in section 6.

We talk about "extending" a pointer structure when all changes are of the form:

- create a new cell in the reference class.
- change this new cell.

Reasoning about extension operations is substantially simpler than that about update operations. This situation is analysed in the following subsection.

4.1.3. Reasoning about extensions

Let $pt = \uparrow t$ be a pointer type; as mentioned above, $\#t$ is the associated reference class. Given a procedure *extend* which changes $\#t$ by extension, then the verifier's semantic checking requires $\#t$ to be declared *var global* to *extend* as in

```
procedure extend(p:pt);
global(var #t);
entry ...
```

If *extend* does not change any cells in the initial collection $\#t_0$, we would like any proposition P that was true for $\#t_0$ still be true for the extended reference class $\#t_f$. But of course, this is not correct. For instance, if P is a proposition about the cardinality of $\#t$ or if it asserts that certain elements are not members of $\#t$.

We have to suitably limit the propositions P we allow to make the above statement true. Consider a pointer p which points to a linked list. Then we can define a function $listrep(p, \#t)$ that maps p and $\#t$ into an abstract list (the concept of representation functions is discussed in section 5 in greater detail).

If we have a proposition P not on $\#t$ but on objects represented in terms of $\#t$, then we have the theorem that

$$P(listrep(p, \#t_0)) => P(listrep(p, \#t_f))$$

because $listrep(p, \#t_0)$ must be independent of those cells which are not yet created and not part of $\#t_0$. This means that all objects defined in terms of the original reference class $\#t_0$ are unchanged after *extend* has been executed. Consequently, if we disallow propositions on reference classes and only allow propositions on abstract objects defined in terms of reference classes the truth of any of these propositions is not effected by *extend*. This allows us to consider the reference class $\#t$ a constant rather than a variable.

This situation can also be explained as follows. Given a program *pgm* that manipulates $\#t$ by pure extensions. Suppose we had an oracle that at the beginning of the execution of *pgm* could guess the final reference class $\#t_f$. In this case we could initialize $\#t$ to $\#t_f$ and no extension operation would ever have to change $\#t$; i.e. $\#t$ could be considered constant throughout *pgm*.

We can make use of the above observation in either of two ways. We can assume a meta theorem that allows to assume reference classes to be constant if they are only changes by extension operations. Alternatively, we can introduce suitable assertions that allow us to use the verifier to deduce relevant instances of this theorem.

We demonstrate the latter alternative in the proof of the parser in chapter IV. The above concepts are easily expressed in our assertion language using predicates *subclass* and *proper*. $subclass(\#t_1, \#t_2)$ is true, if $\#t2$ is an extension of $\#t_1$. *proper* is a predicate on abstract objects defined in terms of reference classes. It is true if such an object is well defined, i.e. if the representation function does not dereference an undefined pointer $\#t \subset p \supset$ where the cell pointed to by p has not been added to $\#t$.

In the verification of the semantic analysis we do not repeat this reasoning, rather we resort to the meta theorem that allows us to consider a reference class unchanged by a procedure if the new reference class is a mere extension of the original reference class.

This reasoning simplifies proofs considerably and lets us concentrate on more relevant things.

4.2. Quantification

In some cases it turns out that the quantifier free assertion language accepted by the Stanford verifier is a severe limitation. There are two principal methods for dealing with quantification.

A simple example illustrates this situation. Let us assume that we have a procedure $p(y)$ for which we need the exit condition $\forall x.R(x, y)$. One solution is to introduce a new predicate symbol $RQ(y)$ with the interpretation $RQ(y) \equiv \forall x.R(x, y)$. This method is general and allows elimination of all existential and universal quantifiers in abitrary situations. The main disadvantage is, that all theorems about R that are only provable in a full first order theory have to be proven outside the verifier and supplied as lemmas.

In many cases a more elegant solution is possible which allows much of the proof to be handled by the verifier. Suppose that for a particular call to p we know that only one certain instance of the exit condition is used for the proof of this call, i.e. $\forall x.R(x, y)$ is particularized to $R(a, y)$ in the proof of this call. In this case we can write a procedure $\dot{p}(x, y)$ with exit condition $R(x, y)$. The additional virtual parameter x is used to provide the correct instance of the exit condition; e.g. if a particular call requires the instance $R(a, y)$ of the exit condition for its proof, then we supply the virtual parameter a in this call. Note, that for this method no new predicates and lemmas are necessary.

This second method does not immediately apply for cases where we actually need the full quantification in the exit assertion. However, the following theorem allows us to verify a procedure in quantifier free logic using virtual parameters and then introduce quantifiers later on. This process is the analogue of the well known $\forall$-introduction in first order logic applied to Hoare's weak program logic.

Suppose we are given a procedure $p(x_1, \ldots, x_n, v_1, \ldots, v_m)$ where all v_i are virtual parameters. Suppose further, that we have a proof for the body S of p with the specifications

$$\{Q(x_1, \ldots, x_n, v_1, \ldots, v_m)\} S \{R(x_1, \ldots, x_n, v_1, \ldots, v_m)\}$$

Clearly, the final values of the var parameters among the x_i do not depend on the initial values of the virtual parameters v_i. Now we can construct a procedure $\dot{p}(x_1, \ldots, x_n)$ with body $\dot{S}$ in which we omit all virtual code.

Theorem [weak $\forall$-introduction] Whenever

$$\{Q(x_1, \ldots, x_n, v_1, \ldots, v_m)\} S \{R(x_1, \ldots, x_n, v_1, \ldots, v_m)\}$$

then it is true that

$$\{x_i = \dot{x}_i\} \dot{S} \{\forall v_i.Q(\dot{x}_1, \ldots, \dot{x}_n, v_1, \ldots, v_m) => R(x_1, \ldots, x_n, v_1, \ldots, v_m)\}$$

Proof Suppose the theorem were false, i.e. let x_i be such that $\dot{S}$ terminates

and assume that there is a tuple v_i with

$$Q(\dot{x}_1, \ldots, \dot{x}_n, v_1, \ldots, v_m) \wedge \neg R(x_1, \ldots, x_n, v_1, \ldots, v_m).$$

Clearly, the entry condition for the original procedure body S is satisfied and S will also terminate for x_i because termination must be independant of the values of virtual variables. But now by

$$Q(x_1, \ldots, x_n, v_1, \ldots, v_m)\{S\}R(x_1, \ldots, x_n, v_1, \ldots, v_m)$$

$R(x_1, \ldots, x_n, v_1, \ldots, v_m)$ must be true which contradicts our assumption. ∎

Of course, we have to be able to express the new exit condition

$$\forall v_i.Q(\dot{x}_1, \ldots, \dot{x}_n, v_1, \ldots, v_m) => R(x_1, \ldots, x_n, v_1, \ldots, v_m)$$

in the assertion language. But this can be done by using the first method and introducing a new predicate symbol, say $QR(\dot{x}_1, \ldots, \dot{x}_n, x_1, \ldots, x_n)$. We still retain the advantage that for the proof of this exit condition we only need the definitions and properties of Q and R and that this proof is done in a quantifier free logic. To make any use of the new exit assertion QR, however, we need the theorem

$$\begin{aligned} &QR(\dot{x}_1, \ldots, \dot{x}_n, x_1, \ldots, x_n) \wedge \\ &Q(\dot{x}_1, \ldots, \dot{x}_n, v_1, \ldots, v_m) \\ &\quad => R(x_1, \ldots, x_n, v_1, \ldots, v_m) \end{aligned}$$

Application of this method is shown in chapter IV when we discuss code generation.

4.3. Computable functions and first order logic

To specify the correctness of our compiler it is necessary to write assertions in first-order logic about functions over domains (cpo's) of Scottery. The following section deals with the question how one can axiomatize such functions and prove useful properties about them.

4.3.1. Standard interpretations

An obvious observation is that the usual axiomatizations of well known theories (e.g. Presburger arithmetic) do not allow corresponding functions from Scott's logic as models (e.g. addition over the flat domain of integers). For example, the sentence $\forall x.x \neq x + 1$ is true for conventional addition but false for the flat integer CPO $N_\perp$ and $+ \in N_\perp \to N_\perp \to N_\perp$.

This problem is relevant for our work since the verifier's prover has built-in decision procedures for certain theories, such as presburger arithmetic, lists, and data structures. Thus the system has $+$ "reserved" as conventional addition and we have to introduce a new symbol to denote addition over the flat integer cpo.

In program proofs we systematically distinguish *concrete* types in the programming language from *abstract* domains in the compiler definition.[7] The relation between abstract and concrete is established through representation functions mapping objects of the program into abstract objects in the definitional language.

For instance, addition on $N_\perp$ can be defined axiomatically as $plus(x, y)$. Using $+$ as addition on integers as it is known to the prover, a possible axiom set is:

$$\begin{aligned} plus(x, \perp) &= \perp \\ plus(\perp, y) &= \perp \\ plus(intrep(x), intrep(y)) &= intrep(x + y) \end{aligned}$$

where *intrep* is a representation function mapping integers into the integer cpo. With these axioms the equation

$$x = plus(x, intrep(1))$$

does not lead to inconsistencies. Rather it only allows to deduce

$$\neg\exists y.x = intrep(y).$$

4.3.2. Higher order functions

In Scott's logic we have to deal with arbitrary high order function (in fact, elements of recursively defined domains can be considered "infinite" order functions [Sc72a]). The question is, how we can describe these objects in first order logic. The straightforward solution is to simply consider functions as values. All operations on functions, including function application have to be introduced as functions in the logic.

In addition we have to be able to talk about least fixed points of these functions. One solution is to axiomatize the ordering *"less defined than"* ($\sqsubseteq$). Given this ordering it can be defined what it means to be the least fixed point etc. The main problem with this approach is, that things become excessively messy and incomprehensible and we should look for a simpler solution.

7.) More on this in section 5

In [CM79] Cartwright and McCarthy propose a *minimization schema* to capture the idea of the least fixed point. But this approach is limited to first order functions and by introducing the predicate isD (to test whether an element is proper) implicitely introduces Scott's ordering ($\sqsubseteq$) for the special case of flat domains.

Here we go a different way. Given a denotational definition with a fixed set of domains. We say that a domain D is *terminal* if there is no other domain $\dot{D}$ defined in terms of D. In particular, elements from a terminal domain can never appear as arguments to functions (otherwise the function would be an element of a domain defined in terms of this terminal domain).

For a given definition we let U be the sum of all non-terminal domains. All objects in U are axiomatized as values (U is the universe of the standard interpretation). Terminal functions from $U \to U$ are first order functions. By the definition of terminal domains there are no higher order functions.

If functions are axiomatized as values we need an operation denoting function application. We use the symbol $\otimes$ for this purpose; i.e. the standard interpretation for $\otimes$ is $\mathcal{T}[\![x \otimes y]\!]\phi = (\mathcal{T}[\![x]\!]\phi)(\mathcal{T}[\![y]\!]\phi)$.

With these tools we can axiomatize non-recursive function definitions in a straightforward way. For example suppose we have a first order term $\dot{T}(x)$ representing the term $T(x)$ in Scottery. A definiton $g = \lambda\, x.T(x)$ is translated into the axiom $g \otimes x = \dot{T}(x)$.

4.3.3. Least fixed points

For the above axiomatization of Scottery to be of any use we have to be able to axiomatize recursively defined functions and least fixed points. Suppose we are given $f = fix(\lambda\, x.T(x))$ and $\dot{T}$ as above. For f we can write the axiom $\dot{T}(f) = f$ which is clearly satisfied by $fix(\lambda x.T(x))$. But the axiom merely asserts that f is a fixed point. It does not require that f be minimal: any fixed point satisfied the above axiom. We say that f is *weakly* axiomatized.

In many cases a weak axiomatization suffices to prove relevant properties of f. Suppose we can give a proof in first order logic of the form

$$f = g \otimes f \vdash P(f).$$

This means that P is true for all interpretations of f satisfying $f = g\ f$. In particular $P(fix\ g)$ is true. Although weak axiomatization is a useful device it is not sufficient in general and many properties P cannot be proven.

Since $fix \in (D \to D) \to D$ is a function we have the obvious property

$$f = g \Rightarrow fix\ f = fix\ g.$$

Again, a useful way of proving properties of fixed points.

In section 6 we discuss yet another way to reason about fixed points. In certain cases it is possible to prove that a program computes a least fixed point without actually introducing the less defined relation $\sqsubseteq$.

5. Representations

In this section we investigate how abstract objects can be represented by suitable data structures in a program. Clearly, in general the problem of data structure selection is very complex and requires human interaction. We are specifically interested in the representation of objects of Scott's computable functions. In this restricted domain several general rules are possible and will be discussed below.

A data structure is either a Pascal type or the product of types, written $T_1, \times \ldots \times T_n$. Since such a tuple could always be defined as a record type we subsequently use the words type and data structure synonymously.

We say that a domain D is represented by a type T by means of r, if $r \in T \to D$ is a "representation" function mapping concrete objects of T into abstract objects in D. To emphasize that such a function has the special use as "representation function" we write $r \in T \leftarrow D$. Semantically $\leftarrow$ is equivalent with $\to$; syntactically $\leftarrow$ has lower precedence than $\to$. A representation function has to be continuous but can otherwise be arbitrary; we require neither surjectivity nor injectivity. Thus, for $d \in D$ there may be none, one or several $t \in T$ such that $r\ t = d$.

Unfortunately, our data types in Pascal are no domains; therefore, whenever we talk about functions defined on data types it is understood that the data type is a flat domain, i.e. has a bottom element added on. But we never use this bottom element to represent an element in a domain since this would not be an "effective" representation.

We now give several examples of types representing domains. More complicated representations will be introduced in the individual program proofs. The representations presented here are very natural ones and will be referred to later without being defined in each case.

5.1. Primitive types

A scalar type T represents the flat domain $T_\perp$ (up to isomorphism). The representation function $Trep \in T \mapsto T_\perp$ is a simple embedding.

For example, we have $truthrep \in boolean \searrow \{TT, FF\}_\perp$ with

$$truthrep(true) = TT, \quad truthrep(false) = FF.$$

A pointer type $P = \uparrow T$ is the set of infinitely many pointers p_i together with the element NIL. Let $D_P = \{p_1, \ldots, p_n, \ldots\}_\perp$; we have $rep_P \in P \searrow D_P$ where $rep_P = \lambda p.$ **if** $p = NIL$ **then** $\perp$ **else** p. In this case we have a representation of $\perp$.

5.2. Complex types

Let $rep_i \in T_i \searrow D_i$ and $T =$ *record* $I_1{:}T_1; \ldots; I_n{:}T_n$ *end*. We define $rep \in T \searrow D$ where $D = D_1 \times \ldots \times D_n$ as

$$rep(x) = \langle rep_1(x.I_1), \ldots, rep_n(x.I_n)\rangle.$$

Had we union types in the verifier's language these would represent sums of domains in the very same manner. Since we do not have union types sum domains are represented as follows. Let $rep_i \in T_i \searrow D_i$ and $D = D_1 + \ldots + D_n$. With

$$T = \textit{record } tag{:}(1..n);\ I_1{:}T_1; \ldots; I_n{:}T_n \textit{ end}$$

we define $rep \in T \searrow D$ as $rep = \lambda x.$ **let** $i = x.tag$ **in** $rep_i(x.I_i){:}D$.

Let T be a scalar type and $rep \in T \searrow T_\perp$ as above, then there is a function rep^{-1} such that $proper\ \epsilon \Rightarrow rep(rep^{-1}\epsilon) = \epsilon$.

Let $rep_1 \in T_1 \searrow D_1$ and $rep_2 \in T_2 \searrow D_2$. With $T = array[T_1]of\ T_2$ and $D = D_1 \to D_2$ we can define a representation $rep \in T \searrow D$. By Pascal rules T_1 must be a scalar type, thus the following is well defined:

$$rep(a) = \lambda\epsilon \in D_1.\ \textbf{if } proper\ \epsilon \textbf{ then } rep_2(a[rep_1^{-1}(\epsilon)]) \textbf{ else } \perp.$$

To make any sense, of course, $rep(a)$ has to be continuous for arbitrary arrays a. This is in fact guaranteed since $rep(a)$ is strict and defined on a flat domain.

Declaring $\uparrow T$, the verifier automatically introduces a type $\#T$, the "reference class" for T. As mentioned earlier a reference class can be thought of as an array indexed by pointers in $\uparrow T$ with T as its component type. Given T, $\uparrow T$, $\#T$ then by the above remarks we have

- $rep \in T \searrow D$,
- $rep_P \in \uparrow T \searrow D_P$.
- $\#rep \in \#T \searrow D_P \dashrightarrow D$, and

In general, a domain need not be represented by a concrete data structure immediately. Rather, it suffices to have a representation for a domain in terms of other domains for which there are concrete representations.

5.3. Recursive domains

Recursively defined domains can also be represented in a straightforward way. Consider the simple example $D = A + (B \times D)$. By the above principle we can construct a representation T with $rep \in T \searrow A + (B \times D_P)$ with $\#rep \in \#T \searrow D_P \to D$, and $rep_P \in \uparrow T \searrow D_P$.

Now consider the data structure $(\uparrow T, \#T)$ consisting of pairs of a pointer $p \in \uparrow T$ and a reference class $r \in \#T$; $(\uparrow T, \#T)$ represents D by means of $recrep \in (\uparrow T \times \#T) \to D$ as follows.

$$\begin{aligned} recrep(p, r) = \textbf{let } x = (\#rep\ r)(rep\ p) \textbf{ in} \\ \textbf{if } x \in A \textbf{ then } x{:}A \textbf{ else} \\ \langle x^{\#1}, recrep(rep_P^{-1}x^{\#2}, r)\rangle \end{aligned}$$

6. Fixed points

6.1. Reasoning about fixed points

At several places we give manual proofs about properties of fixed points. *Sometimes it is more natural to reason about recursion equations than to argue about the fixed point operator (fix).* This is particularly true if "simultaneous" recursion is required. For this reason we introduce the following notation as an alternate for the fixed point operator.

A term of the form

$$\begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ \ldots \\ x_n \end{pmatrix} \leftarrow \begin{pmatrix} T_1(x_1, \ldots, x_n) \\ T_2(x_1, \ldots, x_n) \\ \ldots \\ \ldots \\ T_n(x_1, \ldots, x_n) \end{pmatrix}$$

where T_i are arbitrary terms that may involve x_j's is equivalent to

$$(fix\ \lambda x.\langle T_1(x^{\#1}, \ldots, x^{\#n}), \ldots, T_n(x^{\#1}, \ldots, x^{\#n})\rangle)^{\#1}$$

The following syntactic operations can easily be shown to be sound.

- Row introduction -- If terms T_i contain a term T then several occurrences of T can be replaced by a new variable x_{n+1} if the new row $x_{n+1} \leftarrow T$ is added to the system.[8]

8.) observing some obvious restrictions concerning free and bound variables

- Elimination of redundant rows — If $x_i, i \neq 1$ does not occur in any term except possibly in T_i, then row i can be deleted from the system.
- Elimination of duplicate rows — If two rows i and j, $i, j \neq 1$ are identical up to a renaming of the variables x_i and x_j, then one of these rows, say i, can be deleted if all occurences of x_i are replaced by x_j in the remaining system.
- Substitution — Given a row $x \leftarrow T$, then any number of occurences of x in any T_i may be replaced by T.

We use these rules to prove equalities of the form $fix\ f = fix\ g$ by rewriting both fixed points as recursion equations and transforming them into isomorphic equations. Although this works in many cases it is by no means complete; in general fixed point induction is required.

6.2. Operationalization of fixed points

In the compiler to be written it is necessary to compute least fixed points. For example, in *LS* one can define a type which is a pointer to a type which is a *pointer to a type which is a pointer* ... etc. The semantics of such a recursive type is given as a least fixed point. In order to do typechecking in the compiler we have to compute a representation of this least fixed point. The weak axiomatization introduced in 4.3 is insufficient to prove that a particular computation computes the "least" fixed point. The representation theory presented above allows a way out. Instead of axiomatizing a least fixed point we axiomatize a representation of this fixed point. It turns out that in many cases this representation can be described without the use of fix.

As a simple example consider the domain $D = A + (B \times D)$ for which we defined a representation *recrep* in the previous section. Let $x \in T$ be such that $rep(x) = (a, y)$ and $rep_P^{-1}(y) = p$. With the assignment $p \uparrow\leftarrow x$ we can compute a reference class $\hat{r} = < r, \subset p \supset, x >$. Applying *recrep* we get

$$recrep(p, \hat{r}) = fix(\lambda\, y.(a, y)).$$

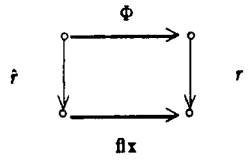
Thus the above theory gives us the possibility to compute the representation of least fixed points without using the least fixed point operator fix directly.

We call this process "operationalization" of fixed points. A similar idea more related to reasoning about programs has been proposed by M. Gordon in [Go75].

The above example may appear to be a special situation; but this is not so; we now prove a more general theorem showing that operationalization can be employed systematically in a variety of cases. Further research has to

determine what the most general situation is that allows for operationalization but the theorem presented below suffices for all cases arising in our work.

In general we proceed as follows: for a domain D be define a representation $r \in T \nwarrow D$ and a representation $\hat{r} \in \hat{T} \nwarrow D \to D$. We then prove that there is an operation Φ on $\hat{T}$, producing an object in T such that $r(\Phi \hat{t}) = fix(\hat{r}\hat{t})$; i.e. the following diagram commutes:



Let D be a recursive domain. We say that a family of function $g_i \in D^{n_i} \to D$ generates D, if every element of D is given by some composition of g_i and $\perp$ (some of the g_i may be constant). This situation obtains in our compiler for many domains, e.g. abstract syntax, modes, and types. Of course, g_i need not all be of the same arity, also, they may take additional parameters from domains not depending on D. What follows adapts easily to either situation.

D can be represented as outlined in the previous section. Let $T = N \times P \times \ldots \times P$ where $P = \uparrow T$; let $\#T$ be the reference class for P. Now D is represented by pairs in $\#T \times P$ by means of $r \in \#T \to P \nwarrow D$ as

$$r\ c\ p = (\ \mathbf{fix}\ \ \lambda h.\lambda x.\ \mathbf{let}\ (i, p_1, \ldots, p_n) = c\ x\ \mathbf{in}\ \mathbf{let}\ y_j = h\ p_j\ \mathbf{in}\ g_i(y_1, \ldots, y_n))p$$

Furthermore we can define a representation for the domain $D \to D$ in terms of triples $\#T \times P \times P$ with $\hat{r} \in \#T \to P \to P \nwarrow D \to D$ as

$$\hat{r}\ c\ p\ q = (\ \mathbf{fix}\ \ \lambda f.\lambda x.\lambda d.\ \mathbf{let}\ (i, p_1, \ldots, p_n) = c\ x\ \mathbf{in}\ \mathbf{let}\ y_j = \mathbf{if}\ x = q\ \mathbf{then}\ c\ p\ \mathbf{else}\ f\ p_j\ d\ \mathbf{in}\ g_i(y_1, \ldots, y_n))p$$

Clearly, not all functions in $D \to D$ can be represented in this way but many useful ones can (recall that we do not require representations to be surjective). We now have the important

Theorem For r and $\hat{r}$ as defined above:

$$r\ (c[c\ p/q])\ p = \mathbf{fix}\ (\hat{r}\ c\ p\ q)$$

Proof We express the least fixed points as recursion equations as introduced in section 6.1. Observing

$$r\,(c[c\ p/q])\ p = (\ \mathbf{fix}\ \lambda h.\lambda x.\ \mathbf{let}\ (i,p_1,\ldots,p_n) = \mathbf{if}\ x = q\ \mathbf{then}\ c\ p\ \mathbf{else}\ c\ x\ \mathbf{in}\ \mathbf{let}\ y_j = h\ p_j\ \mathbf{in}\ g_i(y_1,\ldots,y_n))p$$

the left hand side of the theorem becomes

$$\begin{pmatrix} z_1 \\ h \end{pmatrix} \leftarrow \begin{pmatrix} h\ p \\ \lambda x.\ \mathbf{let}\ (i,p_1,\ldots,p_n) = \mathbf{if}\ x = q\ \mathbf{then}\ c\ p\ \mathbf{else}\ c\ x\ \mathbf{in} \\ \quad \mathbf{let}\ y_j = h\ p_j\ \mathbf{in}\ g_i(y_1,\ldots,y_n) \end{pmatrix}$$

The right hand side may be rewritten as

$$\begin{pmatrix} z_2 \\ f \end{pmatrix} \leftarrow \begin{pmatrix} f\ p\ z_2 \\ \lambda x.\ \mathbf{let}\ (i,p_1,\ldots,p_n) = c\ x\ \mathbf{in} \\ \quad \mathbf{let}\ y_j = \mathbf{if}\ p_j = q\ \mathbf{then}\ z_2\ \mathbf{else}\ f\ p_j\ z_2\ \mathbf{in}\ g_i(y_1,\ldots,y_n) \end{pmatrix}$$

Let us introduce $k = \lambda p.f\ p\ z_1$ in this system yielding

$$\begin{pmatrix} z_2 \\ k \end{pmatrix} \leftarrow \begin{pmatrix} k\ p \\ \lambda x.\ \mathbf{let}\ (i,p_1,\ldots,p_n) = c\ x\ \mathbf{in} \\ \quad \mathbf{let}\ y_j = \mathbf{if}\ p_j = q\ \mathbf{then}\ k\ p\ \mathbf{else}\ k\ p_j\ \mathbf{in}\ g_i(y_1,\ldots,y_n) \end{pmatrix}$$

Now let $t\ p_j = \mathbf{if}\ p_j = q\ \mathbf{then}\ k\ p\ \mathbf{else}\ k\ p_j$ in this system, thus

$$\begin{pmatrix} z_2 \\ t \end{pmatrix} \leftarrow \begin{pmatrix} \mathbf{let}\ (i,p_1,\ldots,p_n) = c\ p\ \mathbf{in}\ \mathbf{let}\ y_j = t\ p_j\ \mathbf{in}\ g_i(y_1,\ldots,y_n) \\ \lambda x.\ \mathbf{let}\ (i,p_1,\ldots,p_n) = \mathbf{if}\ x = q\ \mathbf{then}\ c\ p\ \mathbf{else}\ c\ x\ \mathbf{in} \\ \quad \mathbf{let}\ y_j = t\ p_j\ \mathbf{in}\ g_i(y_1,\ldots,y_n) \end{pmatrix}$$

which proves the equality. ∎

7. Intermediate representation of programs

In our compiler the source program is transformed into various intermediate representations. These are token sequences, syntax trees, and abstract syntax. The concepts of sequences is well understood; this section deals with trees and abstract syntax.

7.1. Trees

7.1.1. Σ-trees on X

[Trees] Let Σ be a finite alphabeth and $a \in \Sigma \mapsto N$ a function assigning an "arity" to each $\sigma \in \Sigma$. Let X be an arbitrary set, then $W_\Sigma(X)$, the set of Σ-trees on X[9] is defined as follows:

- $X \subseteq W_\Sigma(X)$
- if $\tau_1,\ldots,\tau_n \in W_\Sigma(X)$ and $a(\sigma) = n$ then $(\sigma\ \tau_1\ \ldots\ \tau_n)$ is in $W_\Sigma(X)$
- These are all elements in $W_\Sigma(X)$

If X is a domain, then an anologous construction can be devised such that $W_\Sigma(X)$ is a domain too.

7.1.2. Operations on trees

$$\boxed{\mathit{leaves} \in W_\Sigma(X) \mapsto X^*}$$

$$\begin{aligned} \mathit{leaves}\ x &= \langle x \rangle \\ \mathit{leaves}\ (\sigma\ \tau_1\ \ldots\ \tau_n) &= \mathit{leaves}\ \tau_1 \cdot \ldots \cdot \mathit{leaves}\ \tau_n \end{aligned}$$

$$\boxed{\mathit{root} \in W_\Sigma(X) \mapsto \Sigma \bigcup X}$$

$$\begin{aligned} \mathit{root}\ x &= x \\ \mathit{root}\ (\sigma\ \tau_1,\ldots,\tau_n) &= \sigma \end{aligned}$$

7.2. Abstract syntax

The idea of an abstract syntax has first been introduced by McCarthy in [Mc66]. It allows to reason about programs regardless of their external representation (on paper or in a machine). Formally, abstract syntax can be defined as a free algebra; for our purposes abstract syntax is a domain *asyn* of programs.

9.) This pun is intentional: the set of Σ-trees on X is in fact the Σ-word algebra on X; see [Co65].

The domain *asyn* is defined recursively using strict products. Therefore, it is a flat domain of finite trees. *asyn* is a subset of $W_\Sigma(X)$ where X is a set of primitive concepts (like identifiers, numerals etc.) and Σ is a set of constructors to build more complex programs. (To characterize *asyn* more precisely we had to introduce many sorted algebras).

For example, if E and Γ are trees then (*while* E Γ) is an element (tree) in the domain of programs. To increase readability we use an informal notation resembling the external representation of trees. For example, we write

$$\textit{while } E \textit{ do } \Gamma$$

instead of (*while* E Γ).

7.2.1. Constructor and selector functions

Let *zzz* be a name for an element in Σ with $a(zzz) = n$; we agree to use the symbol *mkezzz* as a n-ary function from $(W_\Sigma(X))^n$ to $W_\Sigma(X)$ "constructing" a new tree according to

$$\textit{mkezzz}(\tau_1, \ldots, \tau_n) = (zzz\ \tau_1 \ \ldots \ \tau_n).$$

Furthermore we agree that *deczzz* is the inverse operation to *mkezzz* which returns the list of subtrees of a given tree:

$$\textit{deczzz}(\textit{mkezzz}(\tau_1, \ldots, \tau_n)) = \tau_1 \cdot \ldots \cdot \tau_n.$$

Similarly we use the convention that *iszzz* is a predicate which is true for all trees with root *zzz* and false otherwise.

In later sections we intermix the constructor notation with the above informal one. For example, we use "*while* E *do* Γ *od*" in the formal language definition while we use "*mkewhile*(E, Γ)" in the machine readable lemmas input to the verifier.

7.2.2. Conformity relations

The *conformity relation* :: is used to conveniently access subtrees of a tree. Given $W_\Sigma(X)$ with *zzz* being a name for an element in Σ then

$$E{::}\textit{mkezzz}(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

is true in a context where all y_i are bound and x_i are free if and only if $\exists x_1 \ldots x_n . E = \textit{mkezzz}(x_1, \ldots, x_n, y_1, \ldots, y_m)$. In addition, conformity relations cause free variables to be bound to corresponding components. For example,

$$\textbf{if } E_0{::}\textit{mkezzz}(x_1, \ldots, x_n, y_1, \ldots, y_m) \textbf{ then } E_1 \textbf{ else } E_2$$

means

$$\begin{aligned}&\textbf{if } \textit{iszzz}(E_0) \textbf{ then}\\ &\qquad \textbf{let } x_i = \textit{deczzz}(E)^{\#i} \textbf{ in } E_1\\ &\qquad \textbf{else } E_2\end{aligned}$$

where again x_i are free variables.

Chapter III. Source and target languages

In this chapter we introduce source and target languages *LS* and *LT*. We first give an informal overview over the source language followed by its formal definition. The target language is described in a similar fashion.

1. The source language

One objective of this work is to write a compiler for a realistic and useful language. Since Pascal is a fairly simple but still useful language our source language *LS* is based on Pascal.

A second criterium is that we need a formal definition of *LS*. There exist formal definitions of Pascal of various flavours [HW73, AA77, Te77a]. We build on these, notably Tennent's denotational semantics [Te77a].

However, we made some changes to Pascal. Several language features of Pascal have been omitted. This has quantitative rather than qualitative reasons and was done to keep the task of verifying the compiler manageable. The theory and proof techniques developed in this thesis is general enough to handle all omitted cases.

Some parts of Pascal are poorly defined (see [Ha73, WS77]) and have been changed or extended. Some of the extensions follow naturally from an attempt to formally define the language as has been pointed out in [Te77b].

This section gives an informal overview of *LS* and points out differences to Pascal.

1.1. Data structures

As Habermann [Ha73] points out, the notion of type is not very well defined in Pascal. Aside from his largely informal arguments this also becomes apparent if one tries to give a formal definition of the type concept of Pascal. In fact, in Tennent's definition the Pascal concept is completely abandoned and a structural type equality in the sense of Algol 68 is substituted.

Types in Algol 68 are equal if they are of the same structure. That is two records are equal if they have the same field selectors and equal selctors refer to components of equal type. Two pointer types are equal if they point to equal types.

This concept seems appealing, however, it poses some problems. It is non-trivial to determine the equality of two types. Engelfried [En72] has shown that this is equivalent to determine equivalent states of a finite automaton.[1]

Defining "structural" equality formally is not at all straightforward. In the original report [Wi69] type equality was "defined" in English while the revised report [Wi76] uses a non-intuitive definition in terms of vanWijngaarden predicates. In Tennent's denotational definition of Pascal [Te77a] the Algol 68 style equality is used. One problem in his semantics is that the definition of a monotonic equality on types requires to count the number of types defined in a program which seems rather unnatural. Consider the two types type $p = \uparrow q$; $q = \uparrow p$;. Two types are equal in the Algol 68 sense if they are both pointers to equal types, thus we have $equal(p,q) \equiv equal(q,p)$ which is not well defined. Tennent uses the number n of types defined in an environment and defines $equal(p,q,n) =$ **if** $n = 0$ **then** *true* **else** $equal(q,p,n-1)$.

Pascal's types are very simple to compare; each type declaration defines a new unique type, not equal to any other type. Besides being easy to implement it allows the programmer to take advantage of additional type checking possibilities: one may have two types **array** [1..5] **of** *integer* which are different. The Pascal view of types causes several problems if we consider subranges. A constant of an enumerated type may also be of a subrange type. Operations on enumerated types must be viewed as generic (for all subranges) or elements of a subrange have to be coerced to the supertype before such operations can be applied. Haberman [Ha73] presents various examples involving these problems.

Our solution to types is as follows. Two type declarations define different types as in Pascal. However, subranges do not constitute new types; rather, a subrange type is equal to the supertype of which it is a subrange. Each scalar type in *LS* has a "subrange attribute" which specified the allowable range of values of this type. Variables of type subrange are considered variables of the supertype except that assignments to them causes a runtime check for the bounds.

LS has the predefined types *Integer* and *Boolean*. It allows the definition of enumerated types and subranges [JW76]. Furthermore the user can define ar-

1.) Actually the problem is further complicated by union types which are left out of Engelfried's considerations.

rays with constant bounds, records without variants, and pointer types. Even though variable bounds for arrays would be a reasonalbe extension and make the implementation slightly more interesting they have been omitted. Adding variable bounds means that subranges can no longer be used to define the index range and a new semantic concept has to be introduced in the language.

Notable omissions from Pascal are characters, reals and file types. Also, we do not consider records with variants as their definition is Pascal is unsafe (i.e. violates strong typing).

*LS*allows the definition of constants of type boolean and integer. This concept is a slight generalization of Pascal in that an arbitrary expression is allowed to define a constant provided this expression can be evaluated at compile time.

A major deficiency of Pascal is that there are no constant denotations for arrays and records. In fact, the type concept of Pascal makes it hard to incorporate constant arrays. For example what should the type be of say $(1,2,3,4,5)$. Certainly it is not **array** $[1..5]$ ***of integer***; one possible solution would require constructors for constants explicitly naming the type. Constant denotations for complex types are omitted in *LS*.

1.2. Program structures

Statements of *LS* are those of Pascal with the exception of *for*-loops, *case* and *with*-statements. These are omitted since they are not strictly necessary in the language and their implementation poses no particular problem (possible extensions of our language to include the above features are discussed in chapter V). The *while* and *if* statement are changed to contain a syntactical end symbol which allows *while* bodies and *if* branches to be statement lists without having to write *begin* and *end*.

Pascal's block structure has been extended to allow declarations in inner blocks. *LS*allows jumps as in Pascal; jumps into blocks or statements are illegal. Labels do not have to be declared and may be reused in inner scopes.

Procedures and functions are recursive but not mutually recursive. The reason is that we omit forward declarations of procedures. Alternatively, we could allow procedure and function identifiers to be used before their declaration. Either extension would merely complicate the static semantics; our code generation can handle mutual calls.

Procedures and functions may not be passed as parameters. One problem with procedure parameters is that this mechanism in its full generality requires type checking at runtime. Alternatively one could require the specification of

the parameter types of the formal procedure parameters as in Algol68. For example

procedure *p*(**var** *x*:***integer***; *f*:**function**(***integer***, ***boolean***):***integer***);

Each *LS* program has one anonymous input and one anonymous output file both of type integer. *LS* provides the operations *read*,*eof*, and *write* operating on these files. Opening and closing of files is done automatically at the beginning and end of program execution. Definition, implementation and proof can be extended easily for other predefined functions and procedures; more I/O operations can be added. An extension of the input output facilities to those of Pascal requires the introduction of file types as well as the formal definition of certain component of the operating system environment. For example the semantics of opening a nonexisting file has to be defined.

Expressions in *LS* are those of Pascal with exception of the precedence of operators which is more natural in *LS*.

2. Formal definition of *LS*

2.1. Structure of the formal definition

As outlined earlier a language is defined in several stages: micro syntax, syntax, tree transformation, and semantics. Semantics in turn will be subdivided into static and dynamic semantics. Each of these components defines a mapping; the composition of these mappings defines the meaning of a program. We now define the individual mappings and their composition.

The following domains are common to all components of the compiler. Internally, the definitions of individual components will utilise additional domains.

$c \in Ch$	Characters
$s \in Str = Ch^*$	Strings
Tm	Terminal symols
Vl	Token values
$Tk = Tm \times Vl$	Tokens
St	Syntax trees
Pgm	Abstract programs
E_T	Token errors
E_S	Syntax errors

E_C	Compile time errors
E_R	Runtime errors
$E = E_T + E_S + E_C + E_R$	Errors
$M = N^* \to (N^* + E_R)$	Meaning of programs

Most of these domains are explained in more detail in the corresponding section. For the present discussion their internal structure is irrelevant. It was necessary on this level to define M, the meaning of programs, since this shows that errors can not only occur during compilation but also during execution. The meaning of a program then is a mapping from the input file, N^* into the output file N^* or a possible error.

The language is completely specified by defining the above domains and the following functions:

$scan \in Str \to (Tk^* + E_T)$	Scanner
$parse \in Tk^* \to (St + E_S)$	Parser
$treetr \in St \to Pgm$	Tree transformations
$ssem \in Pgm \to Pgm + E_C$	Static semantics
$dsem \in Pgm \to M + E_R$	Dynamic semantics

[*E*-strict function composition] Let $f \in D_2 \to D_3$ and $g \in D_1 \to D_2 + E$ then *E-strict function composition*, $f \odot g \in D_1 \to D_3 + E$ is defined as

$$f \odot g = \lambda x.\ \textbf{let}\ t = g\ x\ \textbf{in}\ \textbf{if}\ t \in E\ \textbf{then}\ t\ \textbf{else}\ f\ t$$

The meaning of a program is a mapping from character strings into M given as:

$$\boxed{smean \in Str \to M + E}$$

$$smean = dsem \odot ssem \odot treetr \odot parse \odot scan$$

2.2. Denotational semantics

The definitional formalisms used to define micro syntax, syntax, and tree transformations are fairly well known. Denotational semantics, however, is not yet common knowledge. Therefore we review the basic principles at this point. Both static and dynamic semantics are defined using denotational semantics.

2.2.1. General concepts

A denotational language definition is a mapping which assigns a "meaning" to each program.[2] Here program means an abstract program in some abstract syntax as defined above.

The next question of course is, what is the meaning of a program. The method leaves ample room for different meanings. In general, the meaning of a program is a mapping from input data to a set of answers. Answers in turn can be of different flavour. For example, an answer could be an output file or a runtime error. Alternatively, answers could be *true* and *false* merely indicating termination or nontermination.

Note, that a denotational definition is not an interpreter. The meaning of a program is a (function) object which can be reasoned about. The meaning of a program is purely extensional. This means that two programs are considered equal whenever they denote the same function. For example, questions as to the complexity of a program or order of evaluation cannot be answered on the basis of a denotational definition. This in turn has important advantages. In particular in the case of a compiler it leaves ample room for optimizations. As indicated in the introduction, we will prove that source and target program have the same meaning. Thus, sophisticated code optimization could be incorporated in our compiler without having to change the formal language definitions.

One part of a denotational definition is the definition of the underlying domains. By specifying the domains we define the relevant semantic concepts and their relationship. For example, such concepts as types, storable values, locations, answers etc. are specified.

A set of "valuation functions" assigns a meaning to language constructs in terms of the semantic concepts.

The comparison to first order logic may help clarify the situation (see chapter II, section 1.2). Formulas are syntactic objects. An interpretation assigns a meaning (a truthvalue in this case) to each formula. Semantic domains are fairly simple, we only have one universe. But this need not be so, just consider many sorted logical systems.

2.2.2. Semantic concepts of Algol-like languages

Imperative languages such as Pascal and Algol are an outgrowth of the vonNeuman architecture. Therefore their semantics is most naturally described in terms of *locations*, *values* stored in loacations and a *memory*, here called

2.) A program denotes its meaning, hence the name denotational semantics.

store. Let L and V be the domains of locations and values respectively.[3] A store can then be defined as a mapping from L to V, i.e. we write $S = L \to V$; it assigns a value to each location.

Another concept is that of identifiers used in a program. Identifiers name variables, procedures, functions and the like. An *environment* is used to keep track of all the current definitions of identifiers. Considering only variables for the time being, an environment is a mapping which assigns a location to each variable identifiers; we write $U = Id \to L$. In practice environments are more complicated.

What is the meaning (value) of a statement? A naive answer is, that a statement is a mapping from stores to stores; a statement maps the store before its execution into the store after its execution. However, this approach is inadequate to define the meaning of statements that do not normally terminate or alter the "normal" controll flow with a jump. Consider the two statements $\Gamma_1;\Gamma_2$. We would like to describe the meaning of this sequence of two statements as the composition of their individual meaning. Assume that Γ_1 and Γ_2 are defined by a mappings $f_1, f_2 \in S \to S$ respectively. The meaing of $\Gamma_1;\Gamma_2$ would be the mapping $f_2 \circ f_1$. Suppose Γ_1 does not terminate normally but instead produces a runtime error or a jump to another part of the program; our simple model clearly fails: f_2 will not be applied to the result of f_1 since controll never reaches Γ_2 in this case.

To remedy this situation *continuations* have been introduced [SW74]; they are based on an idea derived from "tailrecursion" [Ma71]. A continuation is a function from stores into answers, i.e. $C = S \to A$. At a given point during program execution a continutation specifies the behaviour of the remainder of the program. This behaviour is a mapping of the current store to the final result (answer) of the program. The way to describe the meaning of statements now is as follows. Suppose we know the continuation that obtains after a statement Γ, that is we know what the answer of the program will be once Γ terminated with a given store. Now we can determine the continuation that obtains before the execution of Γ. Thus the meaning of statements will be a mapping from continuations to continuations $(C \to C)$. But note, that the reasoning proceeds backwards, i.e. the meaning of a statement Γ maps the continuation "after" Γ into that "before" Γ.

Using continuations the description of runtime errors and jumps poses no problem at all. Consider the sequence $x \leftarrow a/0;\Gamma$. The meaning of $x \leftarrow a/0$ will be $\lambda\theta\sigma.\mathit{divide\_error}$. That is, whatever continuation θ holds before Γ and

3.) these are further specified in section 6.

after $x \leftarrow a/0$, the continuation before $x \leftarrow a/0$ will be $\lambda\sigma.\mathit{divide\_error}$. I.e. no matter what state we are in when we execute $x \leftarrow a/0$, the answer of the program is *divide_error*.

A similar situation exists for expressions. A naive view is to consider an expression as a mapping from stores to values. Again, this would not capture side effects of expressions, runtime errors, and jumps out of expressions. The solution is the introduction of expression continuations $K = V \to C$. Intuitively, an expression continuation tells us what the final answer of the programm will be given a value (the result of an expression) and a store; it tells us what "to do" with the expression value if we succeed in computing it. The meaning of an expression then is a mapping from expression continuations to continuations. Given the expression E and the expression continuation κ and suppose that E evaluates to ϵ, then the resulting continuation will be $\kappa\epsilon$. On the other hand, if E causes a runtime error, the continuation will be $\lambda\sigma.\mathit{error}$.

In a similar fashion more complex domains are constructed; examples are function and procedure values. For example, a procedure is a mapping in $V^* \to C \to C$, i.e. it takes a list of parameter values and a continuation and produces a new continuation.

Continuations, expression continuations, procedure, and function values all are domains with a similar structure. In the definition of *LS* we introduce the novel concept of generalized continuations which combines all of the above domains into one and leads to some simplification of the language definition (see section 7).

2.2.3. Denotational definition of a machine language

Concepts used to define high level languages can also be used to define machine languages. Since we have no variable names the environment is significantly simpler. In section 8 we describe a stack machine and define appropriate domains. There will be stacks S, memories M, and displays D. Continuations in the target language are mappings of the form $D \to S \to M \to A$. The intuitive meaning of such a continuation is similar to that in the source language case: given a display, a stack, and a memory, the continuation yields the final answer of the program.

2.2.4. Notational issues

A main criticism of denotational semantics is that the notation is too complicated and confusing. We have adopted the notation introduced initially by Scott and Strachey [SS71] and feel that it is very appropriate.

Since the formulas we are dealing with are extremely large, a short notation is imperative for comprehension and overview. Definitions and proofs are more transparent and smaller. As a trade off we have to learn and get familiar with the new notation.

The main points of our notation can be summarized as follows. Function application is written as juxtaposition as in $f\ y$. The semicolon is used to break the normal precedence of function application as in $f\ x; g\ z$. Special parentheses $[\![\ldots]\!]$ are introduced to distinguish syntactic from semantics objects. Attempts are made to have variables named by single letters; we use of roman, greek, and script fonts in upper and lower case.

Mathematics has produced plenty of examples where new short notations have been fruitful. Calculus without $\int$, $\oint$, $\frac{\partial f}{\partial x}$ is unthinkable.

Examples of denotational definitions written in different notations are not very readable, more error prone, and of less help in reasoning about the language [DK79, Pa79, BJ78].

3. Micro syntax

The micro syntax defines a mapping from a sequence of characters into a sequence of tokens. A token is a pair containing syntactic and semantic information.

3.1. Definitional formalism

Micro syntax is defined in two steps. First we define which substrings of the input constitute single tokens. Each of these strings is then mapped into the corresponding syntax semantic pair.

We consider a finite number of classes of tokens; the external representation of elements of each class is given as a regular language L_i. For example an identifier is a string of digits and letters, starting with a letter. Similarly, numbers and single and multi-character delimiters are defined.

Let $\{L_i\}$ be a set of regular languages each defining a class of tokens. We define rules according to which the input string is to be broken into substrings each of which is in one language L_i. The rule is to find the longest leftmost substring s of the input such that $s \in L_i$ for some i. Given $s \in L_i$, a *semantic* function S_i maps the string s into the token represented by s.

For example, L_{id} is the language of identifiers. S_{id} checks if an identifier is a reserved word and treats it differently from ordinary identifiers. Further, an

identifier which is not reserved is encoded as a unique integer by S_{id}. Clearly, to do this the functionality $L_{id} \to Tk$ is too simple for S_{id}. Rather S_{id} needs additional infomation about identifiers that have been scanned previously as well as which codes have already been used to encode identifiers.

We introduce identifier tables

$$h \in H = (L_{id} \to N) \times (N \to \{used, unused\})$$

mapping identifiers into N and specify whether a number in N is *used* or *unused*. Now, the functionality of S_i becomes

$$S_i \in H \to L_i \to (Tk \times H).$$

Let us now describe how a given set of languages L_i and semantics functions S_i are combined to define the scanning function $scan \in Str \to (Tk^* + E_T)$. We define

$$scan = \Psi\ h_0$$

where h_0 is the initial identifier table which contains all predefined identifiers of the language.

$\Psi \in H \to Str \to Tk^*$ deletes initial substrings from the input string until the first character c is the beginning of a token. c is the beginning of a token if $c \in \bigcup a_i$, where a_i is the set of all non-empty initial segments of elements of L_i:

$$a_i = \{u \mid \exists v.u{\cdot}v \in L_i, u \neq \langle\rangle\}$$

Given $c \in \bigcup a_i$ function Φ is applied to recognize the token beginning with c.

$$\begin{aligned}\Psi\ h\ s = {} & \textbf{if } s = \langle\rangle \textbf{ then } \langle\rangle \textbf{ else} \\ & \textbf{if } hd\ s \in \bigcup a_i \textbf{ then } (\Phi\ h\ (hd\ s)\ (tl\ s))^{\#1} \textbf{ else} \\ & \Psi\ h\ (tl\ s)\end{aligned}$$

$\Phi \in H \to Str \to Str \to (Tk^* \times H)$ applied to two strings s_1 and s_2 finds the longest initial substring of $s_1{\cdot}s_2$ which is in some a_i. Φ repeatedly appends the first character of s_2 to s_1 until $s_1 \in \bigcup a_i$ (s_1 is the beginning of a token) and $s_1{\cdot}\langle hd\ s_2\rangle \notin \bigcup a_i$ (i.e. s_1 is the longest possible substring). In this situation Φ returns the token corresponding to s_1 concatenated with the tokens given by the remainder (s_2).

$$\begin{aligned}\Phi\ h\ s_1\ s_2 = {} & \textbf{if } s_2 = \langle\rangle \textbf{ then } \textbf{if } s_1 \in L_i \textbf{ then } (S_i\ h\ s_1)^{\#1} \textbf{ else } error \textbf{ else} \\ & \textbf{if } s_1{\cdot}\langle hd\ s_2\rangle \in \bigcup a_i \textbf{ then } \Phi\ h\ (s_1{\cdot}\langle hd\ s_2\rangle)\ (tl\ s_2) \textbf{ else} \\ & \textbf{if } s_1 \in L_i \textbf{ then } (S_i\ h\ s_1)^{\#1}{\cdot}(\Psi\ (S_i\ h\ s_1)^{\#2}\ s2) \textbf{ else } error\end{aligned}$$

3.2. Micro syntax of LS

To define the micro syntax we have to specify the domains Tm, Vl, languages $\{L_i\}$, semantic functions $\{S_i\}$, and the initial identifier table h_0.

For LS we have 5 regular languages.

L_{id} is the set of all identifiers. The corresponding semantic function S_{id} will detect reserved words and encode all other identifiers as unique integers, using the identifier table h.

L_n is the set of numbers. S_n defines which integer value is denoted by a given string of digits.

L_d is the set of delimiters consisting of just one character. S_d will map each character into its representation as a token.

L_p is the set of all tokens beginning with a period, i.e. {".", ".."}.

L_c is the set of all tokens beginning with a colon, i.e. {":", ":="}.

The complete formal definition of the scanner for LS can be found in appendix 1.

4. Syntax

The syntax of LS is defined by a context free grammar. We extend the definition of grammars to include unique labels for each production. We require that the grammar be unambiguous. In this case the grammar defines a mapping from token sequences Tk^* to parse trees. The production labels become labels of parse trees.

4.1. Labeled context free grammars

Ordinarily a grammar is given by a set of productions. Instead, we specify labeled grammars by a set of lables L together with a mapping P from labels to productions.

$n \in Nt$	Nonterminal symbols
Tm	Terminal symbols
$\lambda \in L$	Labels
$u, v, w \in W = (Nt \bigcup Tm)^*$	Words
$P \in L \mapsto Nt \times W$	Productions

$s_0 \in Nt$ is a distinguished *start symbol*. If $P\,\lambda = (n, w)$ we write

$$[\lambda]\ n ::= w.$$

A grammar G is completely specified by the elements Nt, Tm, L, P, and s_0.

4.1.1. The accepted language

On W we define a binary (infix) relation $\rightarrow \in W \times W$ as follows.

$$u \cdot (n) \cdot v \rightarrow u \cdot w \cdot v \text{ iff } \exists \lambda \in L.P(\lambda) = (n, w).$$

Let $\rightarrow^*$ be the reflexive, transitive closure of $\rightarrow$. For a grammar G we define

$$L(G) = \{w \in Tm^* \mid s_0 \rightarrow^* w\}.$$

Note, that $L(G)$ is a language over strings of terminal symbols.

Recall that the domain of tokens was defined as $Tk = Tm \times Vl$. We extend $L(G)$ for strings of tokens and define:

The language accepted by a labeled context free grammar $L_{Vl}(G)$ is

$$L_{Vl}(G) = L(G) \times^* Vl^*$$

Here $\times^*$ is the obvious componentwise extension of the cartesian product to sequences:

$$S \times^* T = \{(s_1, t_1) \ldots (s_n, t_n) \mid s_1 \ldots s_n \in S, t_1 \ldots t_n \in T\}$$

4.1.2. Parse trees

A tree $s \in S = W_L(Tk)$ is a partial parse tree if

- whenever $\dot{s} = (\lambda\ \tau_1 \ldots \tau_m)$ is a subtree of s, then $P(\lambda) = (n, x_1 \cdot \ldots \cdot x_m)$ and for all $0 \leq i \leq m$ we have either $\tau_i \in Tk$ and $\tau_i^{\#1} = x_i$ or $root\ \tau_i = x_i$.

A partial parse tree s is a parse tree if in addition

- $root\ s = (s_0, w)$ for some w.

4.1.3. The function defined by a labeled grammar

We require that a labeled context free grammar be unambiguous, that is, whenever $s_1, s_2 \in W_L(Tk)$ then $leaves\ s_1 = leaves\ s_2$ implies $s_1 = s_2$.

Each unambiguous labeled context free grammar G defines a "parsing function" $parse \in Tk^* \to St$ as follows.

$parse(w) =$ **if** $w \in L(G)$ **then** s, such that $leaves(s) = w$ **else** $error$

where s is a parse tree for G.

4.2. Syntax of *LS*

Using the above theory, the syntax of *LS* is defined by giving a labeled context free grammar for *LS*. The function *parse* for *LS* is the function defined by this grammar. A labeled context free grammar for *LS* is given in appendix 1. This grammar also satisfies the SLR(1) condition [De71]; we will use this property in the construction of a parser for *LS* which is described in chapter IV.

5. Tree transformations

Tree transformations define a mapping from parse trees into *abstract syntax*. We first define the abstract syntax of *LS* and then specify tree transformations for *LS*.

5.1. Abstract syntax

The abstract syntax *asyn* is the sum of a set of syntactic domains. These domains define syntaxtic entities of *LS*, such as identifiers, expressions, statements and so on. We use a slightly different notation here than we use for semantic domains. For example instead of writing

$$\Theta \in Stm = (Exp \times Exp) + (Exp \times Com) + \ldots$$

we write

$$\Theta :: = E_0 := E_1 \mid \textit{while } E \textit{ do} \Gamma \textit{ od} \mid \ldots$$

for $E_i \in Exp$ and $\Gamma \in Com$. Thus, this notation not only defines the domains but also a particular representation which is used to refer to elements of these domains.

5.1.1. Syntactic domains

$Pgm = Stm$	Programs
$\Omega \in Op$	dyadic operators (not further defined here)

$O \in Mop$	monadic operators (not further defined here)
$I \in Id$	Identifiers (not further defined here)
$N \in Num$	Numerals (not further defined here)
$\Xi \in Pgm$	Programs
$E \in Exp$	Expressions
$\Theta \in Stm$	Statements
$\Gamma \in Com$	Commands
$T \in Typ$	Types
$\Delta \in Dec$	Declarations
$\Delta_c \in Cdef$	Constant Definition
$\Delta_t \in Tdef$	Type definition
$\Delta_v \in Vdec$	Variable declaration
$\Pi \in Par$	Parameters

Expressions

$$E :: = I \mid O\ E \mid E_0\ \Omega\ E_1 \mid E\uparrow \mid I(E^*) \mid E_0[E_1] \mid N \mid E.I$$

Commands

$$\Gamma :: = N{:}\Theta \mid \Theta \mid \Gamma_0;\Gamma_1$$

Statements

$$\Theta :: = E_0 := E_1 \mid \textit{if } E \textit{ then } \Gamma_0 \textit{ else } \Gamma_1 \textit{ fi} \mid \textit{dummy} \mid \textit{while } E \textit{ do } \Gamma \textit{ od} \mid \textit{repeat } \Gamma \textit{ until } E \mid \textit{goto } N \mid I(E^*) \mid \Delta^* \textit{ begin } \Gamma \textit{ end}$$

Declarations

$$\Delta :: = \textit{const } \Delta_c^* \mid \textit{type } \Delta_t^* \mid \textit{var } \Delta_v^* \mid \textit{procedure } I(\Pi^*);\Theta \mid \textit{function } I(\Pi^*){:}T;\Theta$$

Types

$$T :: = I \mid (I_1, \ldots, I_n) \mid E_1..E_2 \mid \textit{array}[T_1]\textit{of } T_2 \mid \textit{record } I_1{:}T_1; I_2{:}T_2; \ldots; I_n{:}T_n \textit{ end} \mid \uparrow I$$

Constant Definitions

$$\Delta_c :: = I = E$$

Type Definitions

$$\Delta_t :: = I = T$$

Variable Declarations

$$\Delta_v :: = I{:}T$$

Parameters

$\Pi ::= I_1{:}I_2 \mid var\ I_1{:}I_2$

In addition to this mathematical notation a machine readable representation is necessary for mechanical proofs. This is provided by constructor and selector functions which are defined in appendix 1.

The domains Id and Num are left undefined. The structure of identifiers is part of the definition of the micro syntax. All that is relevant for the semantics of LS is that we can test identifiers for equality. Similarly, the structure of numerals is irrelevant. We are only interested in the value denoted by a numeral.

5.2. Tree transformations for LS

A tree transformations is defined mapping parse trees into elements of the abstract syntax. The function Ξ is defined recursively on parse trees. One definitional clause is provided for possible node (label of the grammar). For example, the grammar of LS contains the following productions:

$[STMT\_5]$	STMT::= ifsymbol EXPR thensymbol COM fisymbol
$[STMT\_6]$	STMT::= ifsymbol EXPR thensymbol COM elsesymbol COM fisymbol
$[FACT\_1]$	FACT::= lparensymbol EXPR rparensymbol

The corresponding clauses of the tree transformation are

$$\Xi(STMT\_5, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5) = mkeif(\Xi(\tau_2), \Xi(\tau_4), mkestmt(mkedummy))$$
$$\Xi(STMT\_6, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7) = mkeif(\Xi(\tau_2), \Xi(\tau_4), \Xi(\tau_6))$$
$$\Xi(FACT\_1, \tau_1, \tau_2, \tau_3) = \Xi(\tau_2)$$

For example, in the abstract syntax there is no **if** statement without **else** part. The first clause above maps a conditional without matching **else** part into an **if** statement with empty **else** part. The third line shows that parenthesis in expressions are ignored in the abstract syntax. The complete definition of Ξ can be found in appendix 1.

6. Semantics of LS

The semantic definition of LS follows [Te77a]. In particular we use Tennents separation of dynamic and static semantics. The intuition is, that a function is static if it can be evaluated at compile time, dynamic otherwise;

but formally this division is arbitrary, since there is no notions of compile and run time in a formal semantics.

The dynamic semantics of LS is specified on a fairly low level using concepts well known to the compiler constructor. Clearly, a more "abstract" definition of LS is possible. In this case the equivalence with a lower level description such as ours has to be proven as part of the compiler verification. Proof techniques to do this are well established (see for example [MS76]).

Since we are more interested in techniques of specification and verification rather than semantics theories we choose to base the compiler proof on a more detailed definition. This definition can be used directly as the formal basis for the compiler.

The formal definition of LS is presented in appendix 1. In this section we explain some of the interesting points of this definition. We will be informal in our explanations and relate formal objects of the definition to concepts known to compiler constructors.

6.1. Semantic concepts

6.1.1. Semantic domains

In this section we describe the semantic domains used in the definition of the semantics of LS.

T and N are truth values and integers respectively. T is used to model predicates of the descriptive language; the data type *boolean* is not related to T.

$I = \{int_{-\infty}, \ldots, int_{+\infty}\} \subseteq N$ is the set of index values. These are those integers that can be represented in our target machine. The specific values of $int_{+\infty}$ and $int_{-\infty}$ are irrelevant; we assume however, that $0, 1 \in I$.

$Tg = \{int, bool, \nu_1, \ldots, \nu_i, \ldots\}$ is a flat domain of type tags. Type tags are used to create new names for types in LS; i.e. as in Pascal each new type definition creates a unique type. Note, that we cannot simply use the identifier provided in the LS program to name types since there are types without explicit name and scope rules would cause ambiguities.

The concept of locations in LS is fairly complex. First we have a set of absolute locations. These can be thought of as addresses of a real physical memory. Abstract locations are divided into two distinct sets L_d and L_s, called dynamic and static locations respectively. The difference between L_d and L_s is that dynamic objects (created with the *new* statement) are assigned

dynamic locations (on the heap). Variables declared in a block or procedure are allocated static locations (on the stack).

A location in L_d or L_s is able to hold a single object (such as an integer or boolean value). To be able to describe addresses assigned to complex objects such as record and arrays a more complex structure of locations is required. We call those more general locations L-values (Lv). Again, we distinguish static and dynamic L-values, Lv_s and Lv_d. We define

$$v \in Lv_s = L_s + (Id \to Lv_s) + (I \to Lv_s)$$
$$v \in Lv_d = L_d + (Id \to Lv_d) + (I \to Lv_d) + \{NIL\}$$

This means that a static L-value is either a single location (L_s) or the address of a record ($Id \to Lv_s$) or the adddress of an array ($I \to Lv_s$). The address of a record is a mapping which given a component identifier returns the address of this component. Similarly, the address of an array is a mapping from index values into addresses of the components of the array.

In addition to static and dynamic locations we introduce *relative locations* L_r and relative L-values Lv_r. Since procedures and functions in LS can be recursive different incarnations of the same procedure (or function) give raise to different absolute locations for the local variables of this procedure. To describe the meaning of a procedure statically, i.e. independently of a particular call, we assign relative locations and L-values to local variables. At run time these relative locations are mapped into absolute locations by a *memory frame*. We define memory frames as

$$f \in F = L_r \to L_s.$$

Note, that a realtive location is always mapped into a static location since there are no variables that are assigned dynamic locations.

Although memory frames are defined for L_r only they immediately extend to relative L-values as follows. ($\hat{f}$ is the extended function)

$$\hat{f}a = \textbf{if } a \in L_r \textbf{ then } fa \textbf{ else}$$
$$\textbf{if } a \in (I \to Lv_r) \textbf{ then } \lambda\iota.\hat{f}(a\iota) \textbf{ else}$$
$$\textbf{if } a \in (Id \to Lv_r) \textbf{ then } \lambda I.\hat{f}(a[\![I]\!])$$

We assume that there are infinitely many frames $f_i \in F$ and that f_i are totally ordered. The function *succ* gives the successor of a frame, i.e. $succ\ f_i = f_{i+1}$. Further, the ranges of two different frames are distinct, i.e. for all f_i, f_j and a_1 and a_2 we have that $f_i \neq f_j \Rightarrow f_i a_1 \neq f_j a_2$.

Values V are objects that can be stored in memory and can be result of an expression. The only values in LS are index values and pointers, i.e. we have

$V = I + Lv$. In particular there are no array or record values; consequently, arrays and record as whole structures cannot be assigned or passed as value parameters.

A store can be thought of as the memory of an abstract machine; it is a mapping from locations to values. To characterize a computation state completely, a store in LS has two components representing the current input and output file. We define

$$S = (L_s + L_d) \to (V + \{unused\}) \times N^* \times N^*$$

A dynamic location which is unused is mapped into ***unused***. The use of static locations will be recorded in the environment (see below).

An answer of a LS program can either be an output file $\in N^*$ or a runtime error, thus we have $A = N^* + E_R$.

Continuations $C = S \to A$ will be referred to as dynamic continuations. Instead of introducing expression continuations we define dynamic generalized continuations as follows:

$$G_d = C + (V \to G_d)$$

It is easy to see that G_d is isomophic to

$$V \to \ldots \to V \to C$$

for any number of V's; alternatively we can write $V^* \to C$. The use of dynamic generalized continuations will greatly simplify the definition of meaning functions. In addition generalized continuations have an obvious realization in a compiler. Note, that $V^* \to C$ can be interpreted as an expression continuation which takes a "stack" ($\in V^*$) as argument. As can be seen later when we give the definition of meaning functions this stack is exactly the stack used to evaluate expressions in a compiler.

Each call to a procedure requires that new memory be allocated; an executing procedure together with its local memory is called an incarnation of this procedure. Addresses used in the code of a procedure are relative addresses ($\in L_r$). To access the content of a relative locations it has to be converted to an absolute locations using a frame.

The incarnation of the procedure that called the currently executing procedure p is said to be the *dynamic predecessor* of the current incarnation of p.

Every procedure has an associated lexical level, the main program has lexical level 0. If a procedure p with lexical level n is called, then at least one incarnation of all procedures with smaller lexical level ($< n$) which contain

the declaration of p has to exist. If q is such a procedure with lexical level m, $m < n$, then the most recent incarnation of q is said to be the static predecessor of p of level m.

If a variable with relative location a is defined on lexical level m and is accessed in a procedure p with lexical level n, $m < n$, then the frame belonging to the static predecesor of p of level m has to be used to covert a into an absolute location. We introduce a domain of displays to describe this use of frames. The concept of a display is well known to the compiler constructor, see for example [RR64,Gr71].

We define

$$\chi \in X = (B \to X) \times (B \to F) \times X \times B.$$

Here $B = N$ is the domain of lexical levels. A display χ has the following components

- A mapping from lexical levels to displays which gives the static predecessor of the current display.
- A mapping from lexical levels to frames giving the frame corresponding to the static predecessors.
- The dynamic predecessor.
- The lexical level of the currently executed procedure.

At any point the meaning of the remainder of the program depends on the current display. Therefore, we introduce generalized continuations G which depend on a particular display:

$$G = X \to G_d.$$

The meaning of a procedure is a mapping from a list of parameter values and a continuation into the domain of continuations. We could define $V^* \to G \to G$. But this domain is isomorphic to $G \to V^* \to G$ which in turn is isomorphic to $G \to G$. Thus we define $P = G \to G$ as the domain of procedure values. Functions can be modeled by the very same domain. We do not distinguish procedure and function values.

We use two kinds of environments in the definition of *LS*. Static environments U_s contain information about identifiers and numbers relevant for static semantic analysis. Environments U are used to describe dynamic semantics. In

$$U_s = (Id \to Md) \times (Num \to T) \times (Tg \to T)$$

each identifier is mapped into a mode; each number is mapped to a truthvalue indicating whether or not this number is a defined label. Type tags Tg are

mapped into truthvalues to indicate which of the tags have been used to define a type; this is necessary to be able to generate a virgin type tag for the next type declaration.

An evironment maps each variable identifier into a relative L-value, each function and procedure identifier into a procedure value and each label (numeral) into a continuation. In addition for each identifier and label the lexical level of its definition is given. Finally, a component $L_r \to T$ keeps track of used relative locations.

$$\begin{aligned}\rho \in U = &(Id \to Lv_r) \times \\ &(Id \to P) \times \\ &(Id \to B) \times \\ &(Num \to G) \times \\ &(Num \to B) \times \\ &(L_r \to T) \times \\ &B\end{aligned}$$

6.1.2. Types and modes

The domain Ty of types provides an "internal representation" for user defined types. We use a definition similar to that of the abstract syntax. A $\tau \in Ty$ is enclosed in $\lceil \ldots \rceil$, similar to elements of the abstract syntax which are enclosed in $[\![\ldots]\!]$.

Each $\tau \in Ty$ has various fields, one of these is a type tag that uniquely identifies the type. An exception is the *nil* type which is a special pointer type. It is special because *nil* is common to all pointer types and has to be treated differently during type checking. In the following definition products are not strict! The domain Ty contains infinite objects (e.g. recursive types).

$$\tau = \lceil \nu\text{:}sub\text{:}\iota_1\text{:}\iota_2 \rceil \mid \lceil \nu\text{:} \uparrow \tau \rceil \mid \lceil nil \rceil \mid \lceil \nu\text{:}array\text{:}\tau_1\text{:}\tau_2 \rceil \mid \lceil \nu\text{:}record\text{:}(I_1\text{:}\tau_1, \ldots, I_n\text{:}\tau_n) \rceil$$

Each identifier and expression in *LS* is assigned a mode. For example, a mode specifies, whether an object is a constant, a variable, a value, a procedure and so on.

A mode $\mu \in Md$ is enclosed in $\lfloor \ldots \rfloor$; we have the following modes:

- $\lfloor var\text{:}\tau \rfloor$ a variable of type τ.
- $\lfloor vap\text{:}\tau \rfloor$ a var parameter inside a procedure. Its value is a variable of type τ.
- $\lfloor val\text{:}\tau \rfloor$ a value of type τ.
- $\lfloor type\text{:}\tau \rfloor$ a type identifier denoting type τ.

- $\lfloor const:\iota:\tau \rfloor$ a constant of type τ with value ι.
- $\lfloor proc:\mu_1, \ldots, \mu_n \rfloor$ a user defined procedure with parameter modes $\mu_1, \ldots, \mu_n$.
- $\lfloor sproc \rfloor$ a special procedure. A special procedure is a predefined procedure. These are treated differently from user defined procedures to allow for more flexible parameter checking (e.g. predefined procedures can be generic, *new* is an example).
- $\lfloor afun:\mu_1, \ldots, \mu_n:\tau \rfloor$ an active function with parameter modes $\mu_1, \ldots, \mu_n$ and result type τ. A function is active inside its own body; it is passive otherwise. The distinction is relevant since it is possible to assign to an active function but not to a passive function.
- $\lfloor pfun:\mu_1, \ldots, \mu_n:\tau \rfloor$ a passive function.
- $\lfloor sfun \rfloor$ a special function (see remark under special procedure).

6.1.3. Auxiliary functions, static semantics

The following is a list of functions used in defining the statics semantics of *LS* with a brief description.

$distinct \in D^* \to T$ is defined for any flat domain D for which an equality test is defined. It is used to test lists of numbers and identifiers for distinctness.

$typetag \in Ty \to Tg$ selects the type tag of a type.

A type is an index type if it is a subrange (note, that integers are a subrange $int_{-\infty}..int_{+\infty}$). $isindex \in Ty \to T$ returns TT for index types.

A type is returnable if it can be the result type of a function, we have $isreturnable \in Ty \to T$. In *LS* we allow index types, pointers, and the nil type to be returned by functions.

$overlaps, contains \in Ty \to Ty \to T$, for two subranges these functions determine whether they overlap or whether one is contained in the other. They are useful in the definition of assignments: two subranges that do not overlap cannot be assigned; if the right hand side is contained in the left hand side, no runtime check is required.

$union \in Ty \to Ty \to Ty$ constructs the smallest subrange that contains two subrange types.

$type \in Md \to Ty$ selects the type of a mode. For example $type\ \lfloor var:\tau \rfloor = \tau$. *type* is undefined for procedure modes and special functions.

$equal \in Ty \to Ty \to T$ tests two types for equality.

$compatible \in Ty \to Ty \to T$, two types are compatible if there is a possible runtime check that allows these types to be assigned to each other.

For example, equal types are compatible; two subranges that overlap are compatible, a pointer type is compatible with the nil type.

$assignable \in Md \to Md \to T$; mode μ_2 is assignable to mode μ_1 if μ_1 is a variable, if types of μ_1 and μ_2 are compatible, and if the type of μ_2 is returnable. *LS* only allows assignment of objects that are also returnable as function results. Arrays or records cannot be assigned.

If an expression has mode μ_2, $passable \in Md \to Md \to T$ determines if it can be passed to a formal parameter that has mode μ_1. For *var* parameters *LS* requires type equality; in particular, subranges have to match exactly. For value parameters the two modes have to be assignable; i.e. complex data objects cannot be passed as value parameters.

$isvar, isval, \in Md \to T$ check whether a mode is a value or a variable.

$isbool, isint \in Ty \to T$ test if a type is integer or boolean.

$sp \in Id \to Md^* \to T$ checks a call to a special procedure for validity. Given the name of the special procedure and the modes of the actual parameters *sp* returns TT if this call is valid. Note, that this is a very flexible schema; for example a procedure *print* could be made generic for all types; also print could take a variable number of parameters for each call. We assume, however, that *print* has only one integer argument; extensions are easy to add to our compiler.

$sf \in Id \to Md^* \to Md$ is similar to *sp* but checks special functions instead. The result of *sf* is the mode of the object returned by the special function.

$truemode, falsemode \in Md$ are modes of the constants *true* and *false*.

$intconst \in I \to Md$ contructs an integer constant of a given value.

$integer \in Ty$ is the type integer.

$boolean \in Ty$ is the type boolean.

$mkvalmode \in Md \to Md$ makes a value mode with the same type as a given mode.

$w \in \Omega \to Md \to Md \to Md$, $o \in O \to Md \to Md$ are similar to *sf*; they checks binary and unary operators for validity.

6.2. Static semantics

6.2.1. Declarations

A list of declarations is checked by *d* for its correctness. *d* also updates the environment to reflect the new declarations. We have $d \in Dec \to U_e \to U_e$. To

treat constant, type, and var declarations d uses the functions $dc \in Cdef \to U_e \to U_e$, $dt \in Tdef \to U_e \to U_e \to U_e$, and $dv \in Vdec \to U_e \to U_e$.

One of the more interesting points in d is the treatment of recursive types. In d we have the clause

$$d[\![type\Delta_{t1}; \ldots; \Delta_{tn}]\!]\varsigma = fix(dt^*[\![\Delta_{t1}; \ldots; \Delta_{tn})]\!]\varsigma).$$

Note, that dt and dt^*[4] require two environments as argument. To get a feel for the meaning of the fixed point, consider the limit case, where we have $d[\![type\Delta_{t1}; \ldots; \Delta_{tn}]\!]\varsigma = x$ such that $x = dt^*[\![\Delta_{t1}; \ldots; \Delta_{tn})]\!]\varsigma\ x$. If any of the Δ_{ti} contains a pointer type, the reference will be resolved by looking into x instead of ς, whereas ς is used for all other identifiers: they have to be defined in a global scope.

Another interesting case is that of procedure definitions.

$$\begin{aligned} d[\![\textbf{procedure}\ I(\Pi_1, \ldots, \Pi_n); \Theta]\!]\varsigma_0 = \textbf{let}\ &\langle(\mu_1, \ldots, \mu_n), \varsigma_n\rangle ::p^*[\![\Pi_1, \ldots, \Pi_n]\!]\varsigma_0\ \textbf{in} \\ &\textbf{if}\ distinct(i^*[\![\Pi_1, \ldots, \Pi_n]\!] \cdot i[\![\Theta]\!])\ \textbf{then} \\ &\textbf{if}\ g[\![\Theta]\!]\varsigma_n[\lfloor proc(\mu_1, \ldots, \mu_n)\rfloor/I]\ \textbf{then} \\ &\varsigma_0[\lfloor proc(\mu_1, \ldots, \mu_n)\rfloor/I] \end{aligned}$$

p^* is used to determine the modes of the parameters as well as the environment ς_n resulting from ς_0 after entering the parameters with their modes. If all identifiers used as parameters and in the procedure body are distinct and if the procedure body is semantically correct (see definition of g) in ς_n, then the resulting environment is obtained by entering the corresponding procedure mode for I in ς_0.

The remaining cases are straightforward.

6.2.2. Types

$t \in Typ \to U_e \to U_e \to (Ty \times U_e)$ determines the meaning of a type in the program. A given type denotation is evaluated and a representation ($\in Ty$) is constructed. Two points are to be observed here. t changes the environment in two ways. First, an enumeration defines a set of identifiers as constants of this enumeration types; these have to be entered by t in the environment. Furthermore, to represent new types, t uses up type tags which in turn have to be marked as being in use in the new environment.

The second point to observe is that like dt, t too takes two environments as argument. The reason is exactly the same here: t uses the first environment in all cases except to determine the type pointed to by a pointer type declaration.

4.) see appendix 1.

6.2.3. Labels and identifiers

$j \in Com \to Num^*$ returns the list of all labels defined on the top level of a given command.

k returns the list of all procedures and functions defined in a block.

Function i returns all identifiers defined in a given language construct. It is defined for several syntactic domains: $i \in (Par + Stm + Dec + Typ) \to Id^*$.

6.2.4. Expressions

The function $e \in Exp \to U_e \to Md$ evaluates an expression in a given static environment and returns the mode of the expression. In computing this mode e checks if all declaration and type constraints are satisfies. If this is not the case e will return $\perp$. We will explain some typical clauses of the definition.

For an integer constant we have $e[\![N]\!]\varsigma = \lfloor const{:}n{:}\lceil int{:}sub{:}n{:}n\rceil\rfloor$, where n is the value of the constant N (given as the token value). That is, the mode of a constant is the *constantmode* with the value being the value of the constant and the type being the smallest possible subrange containing this constant.

The mode of a binary operator is given by the function w applied to the modes of the operands: $e[\![E_0\ \Omega\ E_1]\!]\varsigma = w[\![\Omega]\!](e[\![E_0]\!]\varsigma, e[\![E_1]\!]\varsigma)$

For an indexing operation to be well typed e has to check that

- the indexed expression is of an array type,
- the type of the indexing expression is compatible with the index type of the array type.

The resulting mode is a var in any case: there are no values or constants of array type. The corresponding definitional clause is:

$$\begin{aligned} e[\![E_0[E_1]]\!]\varsigma = \textbf{if}\ &[\nu{:}array\ \tau_1\ of\ \tau_2]::type\ e[\![E_0]\!]\varsigma\ \textbf{then} \\ &\textbf{if}\ compatible\ \tau_1(type\ e[\![E_1]\!]\varsigma)\ \textbf{then} \\ &\textbf{if}\ isvar\ e[\![E_0]\!]\varsigma\ \textbf{then}\ \lfloor var{:}\tau_2\rfloor\ \textbf{else} \\ &\textbf{if}\ isvap\ e[\![E_0]\!]\varsigma\ \textbf{then}\ \lfloor var{:}\tau_2\rfloor \end{aligned}$$

Other expressions are defined in a straightforward way following the above schema.

6.2.5. Statements

Statements are checked by $g \in Stm \to U_e \to T$ for their semantic validity. The definition is straightforward, for example assignments are defined as

$$g[\![E_1 := E_2]\!]\varsigma = assignable(e[\![E_1]\!]\varsigma)(e[\![E_2]\!]\varsigma)$$

That is, an assignment is legal if the modes of left and right hand side are assignable.

A slightly more complex example are while loops. There are two semantic restrictions: the test must yield a boolean result and the body of the while statement must be valid. For the latter test c is used to evaluate a command. Note, that c is passed a modified environment in which all labels in the while body are defined; this effectively make the body of the while loop a new scope for the purpose of labels. It becomes impossible to jump into a while body.

$$g[\![\textit{while } E \textit{ do } \Gamma \textit{ od}]\!]\varsigma = \textit{isbool}(\textit{type } e[\![E]\!]\varsigma) \wedge \textit{distinct}(j[\![\Gamma]\!]) \wedge c[\![\Gamma]\!]\varsigma[\langle \ldots, \textit{true}, \ldots\rangle / j[\![\Gamma]\!]]$$

6.2.6. Commands

Commands are lists of labeled or unlabeled statements. A list of statements is semantically valid, if all individual statements are valid. Thus for $c \in Com \to U_s \to T$ we have the simple definition:

$$c[\![N{:}\Theta]\!]\varsigma = g[\![\Theta]\!]\varsigma$$
$$c[\![\Theta_1; \Theta_2]\!]\varsigma = c[\![\Theta_1]\!]\varsigma \wedge c[\![\Theta_2]\!]\varsigma$$
$$c[\![\Theta]\!]\varsigma = g[\![\Theta]\!]\varsigma$$

6.3. Dynamic semantics

Meaning functions of the dynamic semantics are denoted by capital script letters $\mathcal{E}$, $\mathcal{D}$ and so on.

6.3.1. Auxiliary functions

By introducing generalized continuations operations on continuations C have to be extended to operate on G_d. Suppose we are given $\gamma \in G_d = V \to \ldots \to V \to S \to A$. We want to define a continuation $\dot{\gamma}$ which reflects a change $f \in S \to S$ to the store. I.e. we want

$$\gamma\epsilon_1 \ldots \epsilon_n(f\,\sigma) = \dot{\gamma}\epsilon_1 \ldots \epsilon_n\sigma$$

The function $\textit{change} \in G_d \to (S \to S) \to G_d$ computes $\dot{\gamma}$ given f and γ.

$\textit{update} \in G \to G$ describes the effect on a continuation of assigning a value to a location:

$$\textit{update}\,\gamma = \lambda\chi\epsilon a.\textit{change}(\gamma\chi)(\lambda\sigma.\sigma[\epsilon/a])$$

$\textit{content} \in G \to G$ decscribes the effect of accessing a location in memory. Let γ be a continuation that takes n arguments $\in V$, then *content* is defined such that

$$\textit{content}\,\chi\gamma = \lambda\chi a\epsilon_2 \ldots \epsilon_n\sigma.(\gamma\chi)(\sigma a)\epsilon_2 \ldots \epsilon_n\sigma$$

$\textit{cond} \in G \to G \to G$ defines a "conditional continuation". Given arguments γ_1 and γ_2 *cond* defines a continuation which takes an argument ϵ and depending on the value of ϵ applies either $\gamma 1$ or γ_2. We define ϵ to be true if $\epsilon \mid I$ is zero.

$$\textit{cond}\gamma_1\gamma_2 = \lambda\chi\epsilon.\ \textbf{if } (\epsilon \mid I) = 0 \textbf{ then } \gamma_1\chi \textbf{ else } \gamma_2\chi$$

$\textit{binop} \in Op \to G \to G$ and $\textit{unop} \in Mop \to G \to G$ define the meaning of binary and unary operators respectively

$Sp \in Id \to Md^* \to G \to G$ defines the meaning of special procedures. In our case we have *print* and *new*. Similarly, $Sf \in Id \to G \to G$ defines the meaning of special functions; in *LS* we have *eof* and *read*.

$\textit{verifys} \in I \to I \to G \to G$ Given two index values and a continuation *verifys* verifies that a value applied to the continuation is in the subrange defined by the two index values. If this condition is not satisfied *verifys* will cause a runtime error. Similarly, $\textit{verifyn} \in G \to G$ will check if a value supplied to a continuation is a pointer not equal to *nil*. *verifyn* is used to ensure that no nil pointer is dereferenced.

$\textit{index} \in G \to G$ and $\textit{select} \in Id \to G \to G$ describe the effect of indexing an array and selecting a component of a record.

Generalized continuation cause one slight problem in the definition of jumps. Suppose we describe the meaning of *goto n* where the continuation following this jump is γ and the continuation of the label n is $\dot{\gamma}$. In certain cases (e.g. if the jump leaves a function) the domains of γ and $\dot{\gamma}$ may differ and require different numbers of arguments $\in V$. Thus, before jumping we have to *adjust* the continuation we jump to. As mentioned above, we may look the V arguments of γ as a stack. It is well known to the compiler constructor that jumps out of functions require adjustment of the runtime stack. This fact is reflected in the formal definition. $\textit{adjust} \in G \to N \to G$ adjusts a given continuation to take n more arguments.

$\textit{args} \in G \to N$ simply determines the number of V arguments of a continuation.

6.3.2. Memory allocation

We define functions to (i) allocate static memory, (ii) dynamic memory on the heap, and (iii) functions to initialize newly allocated memory. In each

case we distinguish objects that occupy one location and complex data objects stored in several locations.

new, *newl*, and *newa* allocate static locations. *new* allocates a single location: $new \in U \to (Lv_r \times U)$. $newl \in Ty \to U \to (Lv_r \times U)$ takes a type as additional argument; it allocates memory for any type. Finally $newa \in I \to I \to Ty \to U \to (Lv_r \times U)$ allocates memory for arrays, given the index values which define the index range and the element type of the array.

The second set of functions is *clear* and *cleara*. These do not allocate memory; rather they initialize newly allocated memory to 0. In *LS* all variables are initialized to 0. Alternatives would be to leave their values arbitrary and have the program behave nondeterministically or to initialize variables to *undefined* and causing a runtime error whenever an uninitialized variable is accessed. The former requires a more complicated definition using power domains [Pl76,Sm78]. The latter was avoided since the resulting semantics cannot efficiently be implemented on most machines; it requires an additional bit in all memory words to indicate initialization.

The third set of functions allocates dynamic locations (on the heap). We have $heap \in S \to (Lv_d \times S)$, $heapa \in I \to I \to Ty \to S \to (Lv_d \times S)$, and $heapl \in Ty \to S \to (Lv_d \times S)$ which are analogous to *new*, *newa*, and *newl* but in addition initialize the alloacated memory.

6.3.3. Declarations

Declarations cause a change in the environment: declared variables are allocated, procedure and function values are entered in the environment. In addition declarations require the execution of initialization code.

Allocation of memory for variables is defined by the functions

$$\mathcal{V} \in Decl \to U_s \to U \to U$$
$$\mathcal{V}^* \in Decl^* \to U_s \to U \to U$$
$$\mathcal{V}_v \in Vdec \to U_s \to U \to U$$

Initialization of variable is described by

$$\mathcal{D} \in Decl \to U_s \to U \to G \to G$$
$$\mathcal{D}^* \in Decl^* \to U_s \to U \to G \to G$$
$$\mathcal{D}_v \in Vdec \to U_s \to U \to G \to G$$
$$\mathcal{D}_v^* \in Vdec^* \to U_s \to U \to G \to G$$

Inside a procedure or function parameters are treated very much like declarations: we allocate memory for each parameter; we initialize each parameter. The difference to declarations is that one location suffices for each parameter;

furthermore, parameters are initialized to the actual parameters rather than 0. Allocation is done by $\mathcal{Q} \in Par \to U_s \to U \to U$ and $\mathcal{Q}^* \in Par^* \to U_s \to U \to U$. Parameters are initialized by $\mathcal{P} \in Par \to U_s \to U \to G \to G$ and $\mathcal{P}^* \in Par^* \to U_s \to U \to E^* \to G \to G$

$\mathcal{F} \in Decl \to U_s \to U \to G^*$ and $\mathcal{F}^* \in Decl^* \to U_s \to U \to G^*$ return the list of procedure and function values defined by a list of declarations. The way these function and procedure values are entered in the environment is described under statements where we discuss the meaning of blocks.

Let us explain the meaning of procedure; functions are treated similarly. We have the definition:

$$\begin{aligned}
\mathcal{F}[\![\textit{procedure } I(\Pi_1,\ldots,\Pi_n);\Theta]\!]\varsigma\rho = &\langle\pi\rangle \\
&\text{where } \pi = \lambda\gamma.\textit{enter}(n+1);\mathcal{P}^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1\rho_1; \\
&\qquad \mathcal{B}[\![\Theta]\!]\varsigma_2\rho_1;\textit{exit } \gamma \\
&\text{where } \varsigma_2 = \varsigma_1[\lfloor \textit{proc}:\mu_1,\ldots,\mu_n\rfloor/I] \\
&\text{where } (\langle\mu_1,\ldots,\mu_n\rangle,\varsigma_1) = \mathcal{P}^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma \\
&\text{where } \rho_1 = \mathcal{Q}^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1(\textit{next } \rho) \\
&\text{where } n = \textit{level } \rho
\end{aligned}$$

ς_2 is the static environment in which formal parameters are bound to their mode and the procedure identifier is bound to the correct procedure mode. The environment ρ_1 has memory allocated for parameters.

The value of the procedure then is given by creating an new frame (*enter*), initializing all parameters, executing the body of the procedure in ς_2 and ρ_2, and discarding the frame for this procedure incarnation (*exit*).

6.3.4. Expressions

Expressions are evaluated by $\mathcal{E}$, $\mathcal{A}$, $\mathcal{L}$, and $\mathcal{R}$. Here, $\mathcal{E}$ evaluates an expression without any coercion. $\mathcal{A}$ takes a mode as additional parameter and coerces the expression to this particular mode. $\mathcal{L}$ and $\mathcal{R}$ essentially are special cases of $\mathcal{A}$ and are used to evaluate expressions on the left hand side and right hand side of an assignment respectively. All definitions are straightforward.

$\mathcal{E}$ does constant evaluation, i.e. if an expression is of constant mode its value is this constant. Although not strictly necessary, we made constant evaluation part of the definition. If it were not, constant evaluation could be an optimization. However, care has to be taken so that for instance the evaluation of 1/0 causes an error at the point defined by the formal semantics. Our definition of *LS* states that this zero division is to be detected at compile time.

$\mathcal{E}$ and $\mathcal{A}$ check for runtime errors such as invalid index and dereferencing of nil-pointers. This is signified by the clauses:

$$\hat{\mathcal{E}}[\![E\uparrow]\!]\varsigma\rho\gamma = \mathcal{R}[\![E]\!]\varsigma\rho; verify_n\,\gamma$$

and

$$\begin{aligned}\mathcal{A}[\![E]\!]\varsigma\mu\rho\gamma = \textbf{if } \mu::[var{:}\tau] \textbf{ then } \mathcal{L}[\![E]\!]\varsigma\rho\gamma \textbf{ else}\\ \textbf{if } \mu::[val{:}\tau] \textbf{ then}\\ (\textbf{ if } \tau::[\nu{:}sub{:}\epsilon_0{:}\epsilon_1] \textbf{ then}\\ \textbf{if } contains\ \tau\tau \textbf{ then } \mathcal{R}[\![E]\!]\varsigma\rho\gamma\\ \textbf{else } \mathcal{R}[\![E]\!]\varsigma\rho; verify_s\ \epsilon_0\epsilon_2\gamma\\ \textbf{else } \mathcal{R}[\![E]\!]\varsigma\rho\gamma)\\ \textbf{where } \tau = type(e[\![E]\!]\varsigma)\end{aligned}$$

6.3.5. Statements

In *LS* we distinguish commands and statements. The reason is to be able to handle jumps properly. In particular we have to rule out that jumps lead into statements.

The meaning of commands is trivially defined as

$$\boxed{C \in Com \to U_s \to U \to G \to G}$$

$$\begin{aligned}C[\![N{:}\Theta]\!]\varsigma\rho\gamma &= B[\![\Theta]\!]\varsigma\rho\gamma\\ C[\![\Theta]\!]\varsigma\rho\gamma &= B[\![\Theta]\!]\varsigma\rho\gamma\\ C[\![\Gamma_0;\Gamma_1]\!]\varsigma\rho\gamma &= C[\![\Gamma_0]\!]\varsigma\rho; C[\![\Gamma_1]\!]\varsigma\rho\gamma\end{aligned}$$

A special function Cl defines the meaning of a command that appears as a component of a composite statement such as the then part of an if-statement. Such a command has to be evaluated in an environment where labels defined in this command are bound to proper continuations. These labels are not bound in an enclosing scope to prohibit jumps into composite statements:

$$\boxed{Cl \in Com \to U_s \to U \to C \to C}$$

$$\begin{aligned}Cl[\![\Gamma]\!]\varsigma\rho\gamma &= C[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma\\ &\quad \textbf{where } \hat{\varsigma} = \varsigma[(\ldots, true, \ldots)/j[\![\Gamma]\!]],\\ &\quad \hat{\rho} = fix\Big(\lambda\dot{\rho}.\rho\big[J[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma/j[\![\Gamma]\!]\big]\Big)\end{aligned}$$

$J \in Com \to U_s \to U \to G \to G^*$ returns a list of continuations corresponding to the labels defined in a command.

$$\begin{aligned}J[\![N{:}\Theta]\!]\varsigma\rho\gamma &= \langle B[\![\Theta]\!]\varsigma\rho\gamma\rangle\\ J[\![\Gamma_1;\Gamma_2]\!]\varsigma\rho\gamma &= J[\![\Gamma_1]\!]\varsigma\rho(C[\![\Gamma_2]\!]\varsigma\rho\gamma)\cdot J[\![\Gamma_2]\!]\varsigma\rho\gamma\\ J[\![\Theta]\!]\varsigma\rho\gamma &= \langle\rangle\end{aligned}$$

Semantics of statements (B) is conventional, however, we should mention two points here. For jumps we have to adjust the continuation as mentioned before. This is done by

$$B[\![goto\ N]\!]\varsigma\rho\gamma = adjust(\rho[\![N]\!])(args\ \gamma - args(\rho[\![N]\!]))$$

The second point of interest are declarations. The meaning of a block is to

- initialize locally declared variables followed by
- execution of the body in the correct environment.

The static environment is determined as for static semantics. In the environment we have to allocate new storage for local variables and bind procedure and function values and label continuations. The latter step requires to construct a fixed point: the continuation at a label depends on continuations of other labels. Similarly, procedure and function values depend on other procedure and function values (for recursive calls) as well as label continuations. Our definition handles mutually recursive procedures.[5]

$$\begin{aligned}B[\![\Delta^*; begin\ \Gamma\ end]\!]\varsigma\rho\gamma &= D^*[\![\Delta^*]\!]\hat{\varsigma}\hat{\rho}; C[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma\\ &\quad \textbf{where } \hat{\varsigma} = d[\![\Delta^*]\!]\varsigma[(\ldots, true, \ldots)/j[\![\Gamma]\!]],\\ &\quad\quad \hat{\rho} = fix\Big(\lambda\dot{\rho}.\ddot{\rho}\big[J[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma/j[\![\Gamma]\!]\big]\big[\mathcal{F}^*[\![\Delta^*]\!]\hat{\varsigma}\hat{\rho}/k[\![\Delta^*]\!]\big]\Big)\\ &\quad \textbf{where } \ddot{\rho} = \mathcal{V}^*[\![\Delta^*]\!]\hat{\varsigma}\rho\end{aligned}$$

7. The target language *LT*

In this section we describe the target language *LT*. First we present the architecture of a hypothetical machine and give an informal description of the language. Strictly speaking it is not necessary to define the machine architecture; the language can be defined independent of a particular implementation.

5.) these are disallowed by the static semantics

Having a concrete machine to talk about will make out explanations easier to understand and transparent.

Following the informal discussion we present a formal denotational definition of LT. Only some interesting parts are discussed here; the complete semantics is presented in appendix 2.

7.1. A hypothetical machine

7.1.1. Design decisions

Ease of translation was one important concern. Therefore, the language contains fairly "high level" concepts. However, it will be simple to verify correct translation of LT in a lower machine language, for instance for a register machine.

The underlying hypothetical machine is a stack machine. The key idea underlying stack architectures is to implement some components of the runtime system for Algol-like languages in hardware [Or73]. The standard runtime organization as well as the machine architecture of the B6700 (a stack machine) [BC71, BC72, BC73] combine conceptually different objects into one stack. This stack is used to (*i*) evaluate expression, (*ii*) allocate static memory, and (*iii*) store administrative information about return addresses of procedures, static and dynamic links of procedure invocations.

In our machine these different objects of the stack are clearly separated which simplifies our proofs. But at the same time it will not be very difficult to encode separate components of our machine in one stack if efficieny dictates so.

7.1.2. Architecture

Our hypothetical machine (see figure 2) consists of the following components.

- An infinite memory capable of storing infinite precision integers.
- A stack used for expression evaluation (potentially infinite)
- a display mechanism (see below).
- one input and one output file.

Our machine knows two kinds of addresses: relative and absolute addresses, both integers. Accessing memory can only be done with absolute addresses. Relative addresses can be converted into absolute ones using information contained in the display (see below).

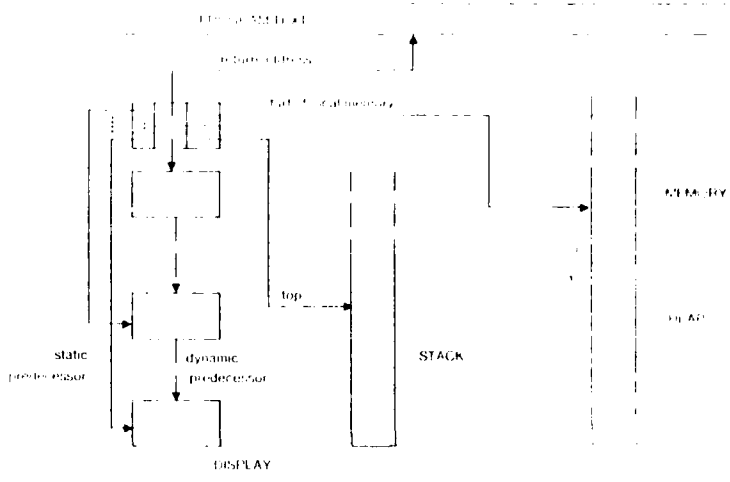


Fig. 2

The memory is divided into two parts, negative and positive addresses. The part with negative addresses will be used as heap to store dynamically allocated objects. Memory with positive addresses is administered in a stack-like fashion; i.e. memory allocated in this part becomes available when the scope in which it was allocated is left.

The display mechanism of our hypothetical machine models the domain X of the source language definition. But the domain D of target display contains more information than X. For each display we have:

- The lexical level of the currently executed procedure.
- A pointer to the displays of all static predecessors.
- The base address of each static predecessor (not shown in the picture of our machine).
- A pointer to the display of the dynamic predecessor.
- The length of the stack upon entry of the current procedure. This information is used by the $JUMP$ instruction if the jump leads out of a procedure or function.
- The return address of the current procedure.

Our design of the display mechanism is not very efficient, but this is not the point of our discussion. More efficient architectures are possible that realize the same semantics.

7.1.3. Instructions

Let us briefly give an informal description of the instructions available on our hypothetical machine. Instructions have varying format; they may have none or several parameters. Generally, all instructions operate on the stack. That is, arguments are obtained from the stack and results are put back on the stack.

- *LIT n* pushs the constant *n* on the stack.
- *LOAD* uses the top of the stack as an address and push the content of the corresponding location.
- *ADDR n m* takes the relative address *n* and the lexical level *m* and compute and absolute address from the current display. The result is pushed on the stack.
- *STORE* pops a value and an absolute address from the stack and updates the address with the value.
- *CALL l n m k* calls the procedure starting at label *l* defined on lexical level *n*. *m* is the number of parameters of this procedure and *k* is the number of static memory locations allocated in the calling environment. *CALL* creates a new display, stores the return address, computes the starting location of the available memory, and marks the level of the stack (current stack length minus the number of parameters).
- *EXIT* exits a procedure or function. This requires to make the dynamic predecessor of the display the new display and branching to the return address of the procedure. The stack remains unchanged, this allows a function result to be pushed on the stack before executing *EXIT*. This result is then available to the callee.
- *NCOND l* inspects the top element of the stack. If it is false ($\neq 0$) then it branchs to label *l*. This instruction assumes that the label is defined on the current lexical level.
- *HOP l* unconditionally branches to label *l*; *l* is assumed to be on the current lexical level.
- *JUMP l n* unconditionally branches to label *l* defined on lexical level *n*. This requires to make the display of level *n* the current display and to

adjust the stack to a length indicated in this display.

- *LABSET n* is a meta instruction and is used to define a label. Its semantics is a no-op.
- *STOP* stops program execution. This is the normal termination; "running" off the end of a list of instructions will cause an error.
- *UNOP n* performs the unary operation number *n* using the top of the stack as argument and pushing the result. We deliberately leave the precise nature of the unary and binary operations undefined. However, they have the same meaning as those in the source language. E.g. $+$ in the source language has the same semantics as addition on the target machine.
- *BINOP n* performs binary operation *n* on the two top elements.

For the sake of simplicity we assume that the machine has standard input output instructions. In practice these instructions would be part of a runtime system.

- *EOF* pushes *true* (0) on the stack if the input file is empty, *false* (1) otherwise.
- *OUTP* prints the top of the stack on the output file.
- *INPT* pushes the first element of the input file on the stack.

7.2. Formal definition of *LT*

As mentioned earlier, no concrete syntax is required since we are not interested in parsing programs written in the target language. Also, we are not interested in checking the static semantics of *LT* programs. We merely define the dynamic semantics. If a program is not well formed, for example if a label is multiply defined, the meaning of this program will be $\perp$.

7.2.1. Abstract syntax

The definition of the abstract syntax is straight forward. We have the domains of labels, numeral, instructions, and programs.

7.2.2. Semantic domains

Integers are as usual; they model values stored in memory as well as addresses. Stacks S are finite lists of integers, $S = V^*$.

Environments map labels into continuations much like in *LS*. Since there are no variables this is the only information in environments.

Displays are defined as

$$\delta \in D = N \to D \times N \to N \times D \times N \times G \times N$$

The components habe the following meaning (in this order)

- Mapping from lexical levels into displays of the static predecessors.
- Mapping from lexical levels to starting addresses of the corresponding memory frame.
- The dynamic predecessor.
- The current lexical level.
- *Continuation to be used upon procedure exit.*
- Stack length of on entry to the current procedure.

The memory $M = (N \to N) \times N^* \times N^* \times N$ is a mapping from addresses to values $(N \to N)$, the input and output file, and the first available location of the heap.

Continuation in LT are mappings from the current machine state into answers. A machine state is characterized by the current display, the stack, and the memory. We curry and write $G = D \to S \to M \to A$.

The domain of answers is identical to answers of LS; an answer is either an output file or a runtime error. $A = N^* + \{eoferror, nostop, \dots\}$

7.2.3. Semantic equations

The valuations for individual instructions are straightforward. We use functions *up* and *down* do create a new display (enter a procedure) and adjust the stack after global jumps.

Labels are handled similar as in LS. The function L returns a list of continuation holding at labels defined in the program. These continuation are bound to the labels in the environment. As in LS we have to construct a fixed point since the continuation at a label will depend on the continuation already bound to other labels in the environment.

Chapter IV. The compiler proof

1. Verifying a compiler

1.1. The compiler

1.1.1. Correctness statement

The meaning of a LS program is given by $smean \in Str \to M + E$. In the previous chapter *smean* is formally defined as

$$smean = dsem \odot ssem \odot treetr \odot parse \odot scan.$$

Similarly, the meaning of a program in the target language LT is given by $tmean \in Cde \to M + E$.

The compiler to be described in this chapter will take a program in LS and produce a sequence of code in LT. Since there is no indeterminacy in our implementation language we can assume that the produced code depends functionally on the input program; i.e. we can look at the compiler as a function $comp \in Str \to Cde$.

The compiler may not terminate, in which case the result of *comp* is $\perp \in Cde$. In particular, we consider error situations as nontermination; i.e. whenever $smean\ x \in E$ then $comp\ x = \perp$. We prove that whenever *comp* x terminates producing code y, then $tmean\ y = smean\ x$. Note, that this correctness statement cannot be expressed by a commutin- diagram.

1.1.2. Structure of the compiler

The compiler is divided into four separate programs executed sequentially. Each program has an input domain and an output domain, such that output and input of two subsequent program match. For example, the scanner produces a sequence of tokens. This sequence is input to the parser which

produces an abstract syntax tree and so on. This is the only communication between the four programs. If we were to include a sophisticated error handling mechanism more information is required, for example one might want to access the identifier table used in the scanner in later stages. Additional information of this kind can easily be communicated without invalidating our verification.

Technical details of how exactly this communication proceeds is left out of our considerations; a concrete implementation may use files or internal data structures to store intermediate forms of the program. Also, a coroutining schema which avoids any intermediate storage is possible.

The division into individual modules is very conventional: we will have a scanner, a parser, a semantic analysis and a code generator. For each program we assume that *input* is the input data and *output* is the output produced. With these conventions the four components of the compiler can be formally specified as follows.

$$\{true\}\ sc\ \{output = scan(input) \mid Tk^*\}$$

$$\{true\}\ pa\ \{output = treetr \odot parse(input) \mid Asyn\}$$

$$\{true\}\ ss\ \{output = ssem(input) \mid Asyn\}$$

$$\{true\}\ cg\ \{tmean(output) = dsem(input)\}$$

The projection into the appropriate domains (e.g. Tk^*) enforces that the output is undefined whenever the formal definition calls for an error. Whenever an error is detected, we will assume that the corresponding program does not terminate and all subsequent programs are not executed.

The compiler is given by executing the above programs sequentially; we write informally *sc; pa; ss; cg*; with the understanding that each program takes the output of its predecessor as its input. Assuming the Hoare's composition rule for statements to be valid for the composition of programs, the compiler will statisfy the following specification.

$$\begin{array}{l}\{true\}\\ \quad sc;\ pa;\ ss;\ cg\\ \quad\quad \{tmean(output) =\\ \quad\quad\ dsem(ssem(treetr \odot parse(scan(input) \mid Tk^*) \mid Asyn) \mid Asyn)\}\end{array}$$

Note, that the error strict functions composition degenerates to normal function composition if we project on non-error elements. Therefore, the exit

condition of the compiler is equivalent to

$$tmean(output) = smean(input).$$

Subsequently, we merely consider the local specification relevant for the individual programs.

1.2. The individual proofs

In the following sections we describe the development of the individual programs. Note, that we do not give program description or a conventional documentation of these programs. Our main interest is the development of the programs, their relation to the problem specification, and their *proofs*.

For each module we describe

- the theorems and lemmas that had to be derived from the specifications to enable an efficient implementation,
- the basic algorithm used in the program and how its correctness can be established from the specifications and derived lemmas and theorems, and
- some relevant implementation details, representations and their effect on the correctness proof.

In addition we will focus on other interesting points of the individual program and will examplify crucial steps in the design methodology. Some typical examples of the program text, underlying theory, and verification conditions are included in the appendix.

The programs we have to deal with are much to large to be handled by the verifier. Consequently, we have to break down the individual programs into smaller pieces that can be verified separately. However, it is necessary to include all routine, type, and variable declarations in each module to present a semantically valid program to the verifier. In total we verified 28 pieces of code comprised of 475 procedures and functions. The final program requires the proof of over 1000 verification conditions. But since each program is developed in a top-down fashion and verification takes place after each refinement step the total number of verification conditions proven during the development of the compiler is much larger.

Frequently, we use virtual data structure, types, procedures, functions, and variables. The verifier requires all these objects to be properly declared. But since virtual objects require no implementation the precise nature of their declaration is irrelevant. Thus the reader may find numerous external procedures, and meaningless type declarations such as

$$\text{type } set = (emptyset, etc).$$

Also real objects may not be refined down to an executable level in which case these too may have nonsensical declarations.

2. A scanner for *LS*

The scanner *sc* has to compute the function $scan \in Ch^* \to (Tk^* + E)$ which is defined in chapter III and appendix 1.

We prove partial correctness of the program *sc* with respect to the following specifications:

$\{input = input_0\}$ *sc* $\{output = scan(input_0)\}$

The defintion of *scan* is part of the logical basis for the verification.

2.1. Underlying theory

2.1.1. A suitable definition

scan is defined by recursive functions Ψ and Φ. These functions specify how a set of regular languages L_i and associated semantic functions S_i are combined to define the micro syntax. For *LS* the languages L_i have two important properties which will allow an efficient implementation of the scanner.

- The initial segments $\{a_i\}$ of the languages $\{L_i\}$ are identical to these languages, i.e.

$$\begin{aligned} L_{id} = a_{id} &= le\ (le + di)^* \\ L_n = a_n &= di\ di^* \\ L_d = a_d &= de \\ L_p = a_p &= pe + (pe\ pe) \\ L_c = a_c &= co + (co\ "="") \end{aligned}$$

- the a_i are disjoint, i.e. $i \neq j \to a_i \bigcap a_j = \emptyset$.

The above properties effectively allow us to recognize one class of tokens, say identifiers, independantly from other token classes. For example, once we have seen a letter in the input this can only be the beginning of an identifier, not a number for example. We make use of this property and define functions Φ_i such that

$$\begin{aligned} \Phi\ h\ s_1\ s_2 = \text{if } & s_1 \in L_{id} \text{ then } \Phi_{id}\ h\ s_1\ s_2 \text{ else} \\ & \text{if } s_1 \in L_n \text{ then } \Phi_n\ h\ s_1\ s_2 \text{ else} \\ & \text{if } s_1 \in L_d \text{ then } \Phi_d\ h\ s_1\ s_2 \text{ else} \\ & \text{if } s_1 \in L_p \text{ then } \Phi_p\ h\ s_1\ s_2 \text{ else} \\ & \text{if } s_1 \in L_c \text{ then } \Phi_c\ h\ s_1\ s_2 \end{aligned}$$

where Φ_i are defined as:

$$\begin{aligned} \Phi_{id}\ h\ s_1\ s_2 = \text{if } & s_2 = \langle\rangle \text{ then } \langle (S_{id}\ h\ s_1)^{\#1} \rangle \text{ else} \\ & \text{if } (hd\ s_2) \in le + di \text{ then } \Phi_{id}\ h\ (s_1 \cdot (hd\ s_2))\ (tl\ s_2) \text{ else} \\ & (S_{id}\ h\ s_1)^{\#1} \cdot (\Psi\ (S_{id}\ h\ s_1)^{\#2}\ s2) \\ \Phi_n\ h\ s_1\ s_2 = \text{if } & s_2 = \langle\rangle \text{ then } (S_n\ h\ s_1)^{\#1} \text{ else} \\ & \text{if } (hd\ s_2) \in di \text{ then } \Phi_n\ h\ (s_1 \cdot (hd\ s_2))\ (tl\ s_2) \text{ else} \\ & (S_n\ h\ s_1)^{\#1} \cdot (\Psi\ (S_n\ h\ s_1)^{\#2}\ s2) \\ \Phi_d\ h\ s_1\ s_2 = \text{if } & s_2 = \langle\rangle \text{ then } (S_d\ h\ s_1)^{\#1} \text{ else} \\ & (S_d\ h\ s_1)^{\#1} \cdot (\Psi\ (S_d\ h\ s_1)^{\#2}\ s2) \\ \Phi_p\ h\ s_1\ s_2 = \text{if } & s_2 = \langle\rangle \text{ then } (S_p\ h\ s_1)^{\#1} \text{ else} \\ & \text{if } (hd\ s_2) \in pe \text{ then} \\ & (S_p\ h\ s_1 \cdot (hd\ s_2))^{\#1} \cdot (\Psi\ (S_p\ h\ s_1 \cdot (hd\ s2))^{\#2}\ (tl\ s2)) \text{ else} \\ & (S_p\ h\ s_1)^{\#1} \cdot (\Psi\ (S_p\ h\ s_1)^{\#2}\ s2) \\ \Phi_c\ h\ s_1\ s_2 = \text{if } & s_2 = \langle\rangle \text{ then } (S_c\ h\ s_1)^{\#1} \text{ else} \\ & \text{if } (hd\ s_2) = "=" \text{ then} \\ & (S_c\ h\ s_1 \cdot (hd\ s_2))^{\#1} \cdot (\Psi\ (S_c\ h\ s_1 \cdot (hd\ s2))^{\#2}\ (tl\ s2)) \text{ else} \\ & (S_c\ h\ s_1)^{\#1} \cdot (\Psi\ (S_c\ h\ s_1)^{\#2}\ s2) \end{aligned}$$

The effect of Φ_i can intuitively be described as follows: given the beginning of a token in L_i then Φ_i completes the scanning of this token. This is conceptually much simpler and allows a more efficient implementation than Φ in the original definition.

We can now change the definition of Ψ to utilise the different Φ_i:

$$\begin{aligned} \Psi\ h\ s = \text{if } & s = \langle\rangle \text{ then } \langle\rangle \text{ else} \\ & \text{if } hd\ s \in a_{id} \text{ then } (\Phi_{id}\ h\ (hd\ s)\ (tl\ s))^{\#1} \text{ else} \\ & \ldots \\ & \text{if } hd\ s \in a_c \text{ then } (\Phi_c\ h\ (hd\ s)\ (tl\ s))^{\#1} \text{ else} \\ & \Psi\ h\ (tl\ s) \end{aligned}$$

Given the simple structure of a_i the tests in the above definition can be simplified to

$$\begin{aligned} hd\ s \in a_{id} &\text{ iff } hd\ s \in le \\ &\ldots\ldots \\ hd\ s \in a_p &\text{ iff } hd\ s = '.' \end{aligned}$$

2.1.2. Axiomatization of concepts

The revised definitions of Ψ and Φ_i are readily expressed in the *rule lan-*

guage accepted by the verifier. For example, the definition

$$\Phi_{id}\ h\ s_1\ s_2 = \text{if } s_2 = \langle\rangle \text{ then } (S_{id}\ h\ s_1)^{\#1} \text{ else}$$
$$\text{if } (hd\ s_2) \in le + di \text{ then } \Phi_{id}\ h\ (s_1 \cdot (hd\ s_2))\ (tl\ s_2) \text{ else}$$
$$(S_{id}\ h\ s_1)^{\#1} \cdot (\Psi\ (S_{id}\ h\ s_1)^{\#2}\ s2)$$

is expressed by the rules

phil1: replace $PHIident(tab, s1, s2)$ where $letter(hd(s2))$ by
$PHIident(tab, concat(s1, list(hd(s2))), tl(s2))$;

phil2: replace $PHIident(tab, s1, s2)$ where $digit(hd(s2))$ by
$PHIident(tab, concat(s1, list(hd(s2))), tl(s2))$;

phil3: replace $PHIident(tab, s1, s2)$ where $\neg digit(hd(s2)) \wedge \neg letter(hd(s2))$ by
$concat(list(Sident(tab, s1)), PSI(STident(tab, s1), s2))$;

phil4: replace $PHIident(tab, s1, null\_sequence)$ by $list(Sident(tab, s1))$;

Similarly we axiomatize other functions of the scanner definition. In addition, we need a theory of sequences. So far we have never made this explict; rather we have taken properties such as $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ and $x \cdot y = x \cdot z \Rightarrow y = z$ for granted. A machine checked proof requires formalization of these properties. The complete set of rules required for the scanner verification is included in appendix 3.

2.2. Basic algorithm

An implementation of recursively defined functions is straightforward and we could simply implement the functions defining the micro syntax. However, a brief look at the resulting complexity shows that this implementation is not very reasonable: the number of recursive calls is equal to the number of characters in the input.

A more efficient implementation is found by using recursion removal techniques. Consider Ψ which calls Φ_i which in turn call Ψ recursively. We will implement Φ_i as procedures which append the token scanned to an output file and then return control to Ψ instead of calling Ψ recursively.

The scanner operates on the following data structures:

- $outfile$ is a file of tokens containing the tokens already scanned.
- $infile$ is the input that remains to be scanned.

The procedure $scan_i$ implementing Φ_i is specified as:

$\{outfile = outfile_0 \wedge infile = infile_0 \wedge hd(infile) = c_0 \wedge c_0 \in L_i\}$
$scan_i$
$\{outfile \cdot \Psi(h, infile) = outfile_0 \cdot \Phi_i(h, hd(infile_0), tl(infile_0))\}$

where h is an identifier table.

Given procedures $scan_i$ as above Ψ can be implemented as a loop. In this loop $scan_i$ (Φ_i) will be called repeatedly such that at any point we have:

$$\Psi\ h_0\ infile_0 = outfile \cdot (\Psi\ h\ infile).$$

In other words, at any time it is true that scanning the remainder (*input*) with the current identifier table (h) and appending this result to the already produced output ($outfile$) is equal to *scan* of the initial input ($\Psi\ h_0\ infile_0$).

The main body of the scanner thus becomes

```
repeat
  if letter(hd(infile)) then scanident else
  if digit(hd(infile)) then scannumber else
  if delim(hd(infile)) then scandelim else
  if hd(infile) = colon then scancol else
  if hd(infile) = period then scanper else
  if ¬eof(infile) then read(infile, c)
until eof(infile)
invariant
Ψ(h0, infile0) = outfile·Ψ(h, infile)
```

The correctness of this high level algorithm follows immediately, given the above the definition of $scan_i$. For example, let us consider a path around the loop where $hd(infile) \in le$. It is to prove that

$\{\Psi(h_0, infile_0) = outfile \cdot \Psi(h, infile) \wedge hd(infile) \in L_{id}\}$
$scanident$
$\{\Psi(h_0, infile_0) = outfile \cdot \Psi(h, infile)\}$

The verifier will generate new variable names denoting the final values of variables changed by *scanident*, in this case $infile_1$, $outfile_1$, and h_1. The verification condition for this path then becomes

$$\Psi(h_0, infile_0) = outfile \cdot \Psi(h, infile) \wedge hd(infile) \in L_{id}$$
$$\Rightarrow hd(infile) = c_0 \wedge c_0 \in L_{id} \wedge$$
$$(outfile_1 \cdot \Psi(h_1, infile_1) = outfile \cdot \Phi_{id}(h, hd(infile), tl(infile))$$
$$\Rightarrow \Psi(h_0, infile_0) = outfile_1 \cdot \Psi(h_1, infile_1))$$

A little simplification shows that this is equivalent to

$$hd(infile) \in L_{id}$$
$$\Rightarrow outfile \cdot \Phi_{id}(h, hd(infile), tl(infile)) = outfile \cdot \Psi(h, infile).$$

This formula is obviously true by the definition of Ψ. Other cases follow in the same way. Thus, at this point we know that our basic design is sound and, if refined correctly, will lead to a correct program.

Implementation of the procedures $scan_i$ poses no problem. Following the definition of Φ_i the programs are immediate using recursion removal. For the complete program of the scanner see appendix 3.

2.3. Implementation details

Now that we decided upon the basic algorithm to be used for the scanner let us look at some of the implementation details.

The abstract program repeatedly refers to the first character of the input stream. To be able to do this in Pascal, we have to read ahead one character and keep the first element of the input stream in a variable. Thus, the remaining input is not given by *infile* but rather as $\langle c \rangle \cdot infile$ for a variable c holding the first element of the character sequence still to be read. However, there is a problem: the input may be empty, $\langle c \rangle \cdot infile$ will never be empty. Tests for the end of file condition have to be recoded. A straightforward solution is to introduce a boolean variable *cvalid* which is true whenever the variable c is part of the input string. The actual assertions in the program have been modified accordingly and distinguish the cases *cvalid* and $\neg cvalid$. The so revised program is shown in figure 3.

Yet another change can be noticed in figure 3. We implement the identifier table h by a data structure *itab* and the representation function *tabrep*. Note, that in the formal definition identifier tables are defined as functions $(\in (L_{id} \to N) \times (N \to \{used, unused\}))$. In the implementation these functions are represented as data objects. In languages like Lisp where it is possible to dynamically redefine functions (compute on functions) identifier tables could be implemented as functions although this may not be very efficient. We write $apply(h, s)$ for function application, i.e. $apply(h, s)$ in the program documentation corresponds to $h\ s$ in the formal definition.

itab is a record with three components:

- an array of strings which contains identifiers,
- an array of encodings for these identifiers, and
- an integer index pointing to the top element allocated in both arrays.

For the complete definition of this data structure the reader may refer to appendix 3 which contains the complete program text with formal documentation for the scanner.


```
begin
  read(infile, c);
  cvalid ← true;

  repeat
    if letter(c) then scanident else
    if digit(c) then scannumber else
    if delim(c) then scandelim else
    if c = colon then scancol else
    if c = period then scanper else
    if eof(infile) then cvalid ← false else read(infile, c)
  until not cvalid
  invariant
  (((cvalid ∧ PSI(tabrep(itab0), infile0) =
    concat(outfile, PSI(tabrep(itab), concat(list(c), infile))))
  ∨(¬cvalid ∧ PSI(tabrep(itab0), infile0) = outfile)) ∧ (cvalid ∨ eof(infile)))
    ∧isitable(itab);

end
```

Fig. 3

A similar refinement is carried out for the procedures $scan_{ij}$ for the complete program see appendix 3.

3. A parser for *LS*

For the scanner we had a formal definition which was readily transformed into a concrete program. The situation is different in the case of the parser. The parsing function is defined very indirectly, i.e. the result of parsing a program is the unique parse tree if it exists. This parse tree is defined axiomatically; we have to find an algorithm that allows us to compute this tree effectively.

The theory of parsing is well understood and we can resort to existing results in literature [AJ74, AE73, AU72, De71, Kn65]. Our compiler uses an LR-parser; the necessary theory of LR-parsing is introduced in the next subsection.

3.1. LR theory

The theory of LR-parsing allows the generation of a *parsing table* which

can be used to drive a parser. Different algorithms for generating tables exist which vary in the size of the generated tables and the class of language that can be handled. For all these methods the parsing algorithms using the tables are identical. Below, we describe the part of LR-theory necessary for the verification of the parser[1]. Although the theory of LR-parsing is well known for some time, a machine checked correctness proof of an LR-parser has not been given.

3.1.1. LR-parsing tables

Let G be a labeled context free grammar satisfying the following conditions:

- There is only one label $\lambda_0 \in L$ such that $(P\lambda_0)^{\#1} = s_0$.
- $P\lambda_0 = \langle s_0, \langle n, t\rangle\rangle$ where $n \in Nt$ and $t \in Tm$ such that t does not occur in any other production.

An LR-parsing table for G is a mapping $slr \in St \to (Nt + Tm) \to Ac$ where Nt and Tm are the set of nonterminals and terminals of G and St is a finite set of states with a distinguished initial state $z \in St$. The set Ac is of the form

$$\begin{aligned} Ac = &\{error\} + \\ &\{shift\} \times St + \\ &\{accept, reduce, shiftreduce\} \times L \end{aligned}$$

Let us first give an informal overview showing how these parsing tables are used in the parser. We will then formalize these ideas and show how the correctness of a LR-parser can be proven.

The basic LR-parsing algorithm uses two stacks, *statestack*, a stack of states, and *ntstack*, a stack of terminals and nonterminals. An input sequence *input* contains the string of tokens to be parsed. The algorithm starts with an empty stack *ntstack* and with *statestack* containing the initial state z.

The algorithm proceeds by repeatedly performing "parsing actions". Let $s = top(statestack)$ and $c = hd(input)$, then the parsing actions to be performed can be described as follows:

- If $slr(s,c) = error$ print an error message.
- If $slr(s,c) = \langle shift, \acute{s}\rangle$ push $\acute{s}$ on *statestack*, push c on *ntstack*, and remove c from the input.
- If $slr(s,c) = \langle reduce, \lambda\rangle$ and $P\,\lambda = < n, t_1 \ldots t_m >$ then remove m items from *statestack* and *ntstack* and make n the first symbol of the input stream (i.e. consider $\langle n\rangle \cdot input$ next).
- If $slr(s,c) = \langle shiftreduce, \lambda\rangle$ and $P\,\lambda = < n, t_1 \ldots t_m >$ then remove $m-1$ items from *statestack* and *ntstack*, delete c from the input, and consider n the next symbol in the input stream.
- If $slr(s,c) = \langle accept, \lambda\rangle$ then the input is parsed correctly.

Successively performing the above steps results in a bottom up parsing of the input. Characters of the input string are shifted on *ntstack* until the right hand side of a production λ is on top of *ntstack*. Then production λ is applied, that is, the right hand side is replaced by the left hand side. In addition to the above steps the appropriate parse tree has to be constructed; we will discuss this later.

Our informal characterization of the parsing tables is, of course, insufficient for a formal proof. Why should a parser constructed in the above way work? We now present a formal chracterization of LR-parsing tables which then enables us to state and prove the correctness of the parser.

With $W = (Nt + Tm)^*$ we define a relation between sequences of states and sequences of terminals and nonterminals. $slrrel \subseteq St^* \times W$ as follows:

$$slrrel(\langle z\rangle, ())$$
$$slrrel(u\cdot\langle s_1\rangle\cdot\langle s_2\rangle, w\cdot\langle x\rangle) \text{ iff } slrrel(u\cdot\langle s_1\rangle, w) \wedge slr(s_1, x) = \langle shift, s_2\rangle$$

$slrrel(u, v)$ is true, if u is of length $n+1$, v is of length n and if the $i+1$-st element of u is determined by slr applied to the i-th element of u and the i-th element of v. In the proof of the parser we show that at any time the relation $slrrel(statestack, ntstack)$ holds.

A LR-parsing table has the following relevant properties which enable the construction of a parser. These properties can easily be shown based on the construction algorithm, see for example [De71].

Whenever slr calls for a reduce action with production λ then the right hand side of production λ is a suffix of v

$$\begin{aligned} &\text{if } slrrel(u\cdot\langle s\rangle, v) \wedge slr(s,x) = \langle reduce, \lambda\rangle \\ &\quad \text{then } v = w\cdot(P\lambda)^{\#2} \text{ for some } w \end{aligned}$$

Whenever $slr(s,x)$ calls for a shift-reduce action with production λ then some suffix of v followed by x constitutes the right hand side of λ

$$\begin{aligned} &\text{if } slrrel(u\cdot\langle s\rangle, v) \wedge slr(s,x) = \langle shiftreduce, \lambda\rangle \\ &\quad \text{then } v\cdot\langle x\rangle = w\cdot(P\lambda)^{\#2} \text{ for some } w \end{aligned}$$

An accept action acts like a shift-reduce action.

$$\begin{aligned} &\text{if } slrrel(u\cdot\langle s\rangle, v) \wedge slr(s,x) = \langle accept, \lambda\rangle \\ &\quad \text{then } \langle s_0, v\cdot\langle x\rangle\rangle = P\lambda \end{aligned}$$

1.) The tables actually used in the compiler were generated with and SLR generator

Whenever $slr(s,z)$ indicates an error, then v concatenated with z is not a prefix of any syntactically valid program.

$$\text{if } slrrel(u\cdot\langle s\rangle, v) \wedge slr(s,z) = error \text{ then } \forall w.(v\cdot\langle z\rangle\cdot w \notin L(G))$$

3.1.2. The LR-parsing algorithm

We now describe how to use parsing tables introduced above to parse a string and construct a parse tree. The parse tree for an input string is defined in chapter III. It should not be confused with the abstract syntax tree, which is an element of the domain *asyn* of the abstract syntax of *LS*.

Let Tr be the set of all partial parse trees for G. We define a relation $isderiv \subseteq Tr^* \times Tk^*$ which holds for a forest $r_1,\ldots,r_n$ and a string of tokens w if and only if the leaves of trees r_i concatenated yield w. Recursively we define this as:

$$isderiv(\langle\rangle,\langle\rangle)$$
$$isderiv(u,v) \wedge leaves(s) = w \Rightarrow isderiv(u\cdot\langle s\rangle, v\cdot w)$$

Let $input_0$ be the initial input to the parser, then p is a parse tree for the string $input_0$ if

$$isderiv(\langle p\rangle, input0) \wedge root(p) = s_0$$

where s_0 is the startsymbol of grammar G. Thus, $p = parse(input_0)$ according to the definition of *parse* in chapter III.

In addition to the *ntstack* and *statestack* the parser uses a third stack *treestack* which is used to construct the parse tree.

The program maintains the following invariant for these stacks:

$$isderiv(treestack\cdot mketree^*(input), input_0) \wedge$$
$$slrrel(statestack, ntstack) \wedge$$
$$root^*(treestack) = ntstack$$

Here *mketree* constructs a singleton tree: $mketree = \lambda x.x{:}Tr$. The algorithm proceeds by performing appropriate shift and reduce actions until the input string is accepted.

The basic structure of the algorithm is shown in figure 4. The construction of the parse tree is merely indicated by comments. *treestack* is used to build parse trees in the following way: whenever a token is pushed on *ntstack*, a singleton tree consisting of just this token is pushed on *treestack*. Then, whenever the parser does a reduction step the partial parse trees corresponding

to the right hand side of the production are on *treestack*. These are used to construct a new tree which is pushed on *treestack*. Similarly we proceed for shift–reduce actions.

Based on the LR–theory above the parsing algorithm can be verified (figure 4 just indicates the principle but is not correct Pascal).

3.1.3. Axiomatization

The necessary theories are readily expressed in the verifier's rule language. For example, the LR–theory is given by

```
%definition of isderiv %
deriv1: infer isderiv(mkeforest(z), z);
deriv2: infer isderiv(concat(concat(u, list(nd)), v), w) from
            isderiv(concat(concat(u, ts), v), w) ∧ roots(ts) = rhs(production(nd));
deriv3: infer isderiv(concat(list(nd), v), w) from
            isderiv(v, w) ∧ null_sequence = rhs(production(nd));
deriv4: infer isderiv(list(mketree(p, n, u, r)), w) from
            isderiv(u, w) ∧ roots(u) = rhs(p);

%definition of slrrel %
slrr1: infer slrrel(list(initial_state), null_sequence);
slrr2: infer slrrel(concat(u, list(s)), concat(v, list(n))) from
          slrrel(u, v) ∧ s = slr(hd(last(1, u)), n).state;
%consequence of 1 and 2 above (requires induction to prove) %
slrr3: infer slrrel(remain(n, s), remain(n, nt));

%properties of LR-parsing tables %
lr1: whenever rhs(slr(s, z).prod)
     from slrrel(st, nt) ∧ slr(s, z).skind = reduce ∧ list(s) = last(1, st)
     infer last(length(slr(s, z).prod), nt) = rhs(slr(s, z).prod);
lr2: whenever rhs(slr(s, z).prod)
     from slrrel(st, nt) ∧ slr(s, z).skind = shiftreduce ∧ s = hd(last(1, st))
     infer concat(last(length(slr(s, z).prod) − 1, nt), list(z)) = rhs(slr(s, z).prod);
lr3: from slrrel(st, nt) ∧ slr(hd(last(1, st)), z.syn).skind = accept
     infer concat(nt, list(z.syn)) = rhs(slr(hd(last(1, st)), z.syn).prod) ∧
       len(nt) = 1 ∧
       length(slr(hd(last(1, st)), z.syn).prod) = 2 ∧
       lhs(slr(hd(last(1, st)), z.syn).prod) = startsymbol ∧
       z = eof_symbol;
```

```
begin
 push(statestack, initial_state);
 repeat
   act ← slr(top(statestack), hd(input));
   if act.skind = error then errmsg else
   if act.skind = shift then
   begin
     push(statestack, act.state);
     push(ntstack, hd(input));
     input ← tl(input);
     'build a singleton tree'
   end else
   if act.skind ≠ accept then begin
     if act.skind = reduce then
     begin
       n ← length(act.prod);
       nt ← lhs(act.prod);
       nlst ← npop(ntstack, n);
       npop(statestackptr, n);
       'build a tree according to the production nt , nlst'
       act ← slr(top(statestackptr), nt);
     end else input ← tl(input);
     while act.skind = shiftreduce do
     begin
       n ← length(act.prod);
       'note, only n-1 items are on the stack!'
       nlst ← npop(ntstack, n − 1);
       npop(statestackptr, n − 1);
       nt ← lhs(act.prod);
       'build a tree'
       act ← slr(top(statestack), nt);
     end ;
     push(ntstack, nt);
     push(statestackptr, act.state);
   end ;
 until act.skind = accept
 'accept the input'
end .
```

Fig. 4

3.2. Tree transformations

As mentioned earlier the parser will also perform the tree transformations. This section first outlines a slightly modified definition of the tree transformation better suited for an implementation. Next, we show how the transformation step is integrated in the parser.

3.2.1. Building abstract syntax trees

The tree transformation function Ξ maps $W_L(Tk)$ into *Asyn*. Since Ξ is defined recursively, the most natural implementation is to first construct a parse tree and then apply the Ξ. However, the particular structure of the tree transformations for *LS* allows a much more efficient implementation.

We define a function $trtr \in W_L(Asyn) \rightarrow Asyn$ such that the following holds:

$$\Xi(\lambda\ \tau_1 \ldots \tau_n) = trtr(\lambda\ \Xi(\tau_1) \ldots \Xi(\tau_n))$$

Of course, it is not always possible to construct such a function *trtr* but in our case the structure of Ξ is simple enough.

The importance of the definition of *trtr* is, that it allows us to compute $\Xi(\lambda\ \tau_1 \ldots \tau_n)$ without knowing the actual subtrees τ_i. Instead, it is sufficient to know the already transformed subtrees $\Xi(\tau_i)$. This enables us to construct the abstract syntax *bottom up*, simultaneously with the construction of the parse tree.

To construct the abstract syntax tree during parsing we add a fourth stack *aststack* to the parser. This new stack will be related to *treestack* such that at any time we have $\Xi^*(treestack) = aststack$.

We maintain *aststack* as follows: whenever a singleton tree τ is pushed onto *treestack* we push $\Xi(\tau)$ on *aststack*. Consider a reduction step that takes $\tau_1, \ldots, \tau_n$ from *treestack* and pushes $(\lambda\ \tau_1 \ldots \tau_n)$ on *treestack*. For this reduction we remove $a_1, \ldots, a_n$ from *aststack* and push $trtr(\lambda\ a_1 \ldots a_n)$ on *aststack*. By $\Xi^*(treestack) = aststack$ we have $\Xi^*\langle\tau_1 \ldots \tau_n\rangle) = \langle a_1 \ldots a_n\rangle$. And by definition of *trtr* the invariant $\Xi^*(treestack) = aststack$ is preserved.

Adding the construction of the abstract syntax to the parser reveals that *treestack* as well as *ntstack* are never really needed except in the formal documentation. Consequently, both these stacks will be virtual data structures that require no implementation.

3.3. Refinement

3.3.1. Development cycle

The actual development of the parser took 10 refinement steps. The intermediate versions of the program can be roughly characterized as follows.

- bare bone version; parsing actions are assumed to be external procedures. Stack operations are abstract, so is the input file. There is no output as yet.
- same as previous version but parsing actions are implemented in-line.
- *aststack* is introduced to build an abstract syntax tree.
- a special variable to hold the lookahead is introduced.
- A type *tree* (with pointers) has been defined, *aststack* is implemented as array of trees. However, tree-building operation are still external.
- *statestack* is implemented; now all stacks are implemented, recall that *ntstack* and *treestack* are virtual.
- Reference classes have been introduced together with the necessary documentation to prove that existing trees are unchanged while new ones are being built.
- Auxilliary functions are verified. This step required the change of some entry and exit assertions which in turn called for a reverification of main program with strengthened invariants.
- Routines that build the abstract trees are implemented and verified.

3.3.2. Representation

The parser actually does not parse the string contained on the input file but instead parses $input \cdot \langle eof\_symbol \rangle$ where *eof_symbol* is a new unique token. This token is automatically supplied by the input routine *get* when the input file is empty.

Recall that the grammar has to have the property that $P\lambda_0 = \langle s_0, \langle n, t \rangle \rangle$ such that t does not occur in any other production. Thus, given an arbitrary grammar with start symbol n we will be parsing an augmented grammar with start symbol s_0 and the new production $\langle s_0, \langle n, eof\_symbol \rangle \rangle$ which then is guaranteed to satisfy the condition required in 3.1.1.

The predicate $uniqueeof(x)$ is true for a sequence x if *eof_symbol* only occurs as the last element and not somewhere in the middle of x.

The input sequence of tokens is represented as a pair $\langle l, input \rangle$ where l is the lookahead token and *input* is the remaining input file. The representation function is defined as

$$seqrep(eof\_symbol, input) = \langle eof\_symbol \rangle$$
$$seqrep(l, input) = \langle l \rangle \cdot input \cdot \langle eof\_symbol \rangle \text{ where } l \neq eof\_symbol$$

An element of the abstract syntax is represented as a pair of a reference class and pointer. It is based on the principles for representing recursively defined domains (see chapter II); a complete definition of the corresponding representation function *synrep* is contained in appendix 4.

We have to define representations for two stacks. In either case we chose an array together with a starting index and a length. For efficiency reasons we do not use the starting index 1. Since we have to implement the operation *npop* which partitions a stack into two sequences, i.e.

$$npop(s, n) = \langle s_1, s_2 \rangle$$

such that $s_1 \cdot s_2 = s$ and $length(s_2) = n$ it is convinient to store both sequences in the same array with different starting indices. The representation functions are as followsns

$$astseqrep(rc, ar, s, 0) = \langle\rangle$$
$$astseqrep(rc, ra, s, f) = astseqrep(rc, ra, s, f-1) \cdot \langle synrep(rc, ra[s+f-1]) \rangle$$
$$stateseqrep(ar, s, 0) = \langle\rangle$$
$$stateseqrep(ar, s, f) = stateseqrep(ar, s, f-1) \cdot \langle ar[s+f-1] \rangle$$

In addition to the standard operations on sequences we introduce the function $last(n, s)$ denoting the last n elements of sequence s. Similarly, we have the complementary function $remain(n, s)$ such that

$$remain(n, s) \cdot last(n, s) = s.$$

Yet another point should be mentioned. We are only interested in the case where the program terminates normally. We do not particularly care what happens after an error has been detected except that we require that an error procedure is called and the error be reported to the user. Thus we define a procedure error as

```
procedure error;
entry true; exit false; external;
```

The exit assertion *false* guarantees that the verification condition for any path containing a call to *error* will be true.

3.3.3. Reference classes and pointer operations

In chapter II we presented a theorem which simplifies the reasoning about extension operations on pointers. Suitable instances of this theorem can be added to the logical basis of a proof and we will do this in later cases. However, it is also possible to introduce suitable concepts in the assertion language and let the verifier prove necessary instances of the theorem. We demonstrate this in the proof of the parser.

Given a data type T and a representation function $rep \in T \leftrightarrow D$. In chapter II we defined domains represented by a pointer to T and the associated reference class $\#T$ as

- $rep_P \in \uparrow T \leftrightarrow D_P$ for a flat domain D_P and
- $\#rep \in \#T \leftrightarrow (D_P \to D)$.

The mapping represented by a reference class is partial, i.e. we have

$$\#rep(r)\ rep_P(p) = \bot$$

for pointers p which have not been added to r by means of the extension operation $r \bigcup \{p\}$.

We define the predicate *subclass* as follows

$$subclass(r_1, r_2) \text{ iff } \#rep(r_1) \sqsubseteq \#rep(r_2)$$

i.e. $subclass(r_1, r_2)$ is true if and only if r_2 is an extension of r_1.

The recursive domain *asyn* of abstract syntax is represented by a pointer and a reference class as outlined in chapter II; we have

$$synrep \in \#T \times \uparrow T \to asyn$$

for a suitable record type T (for the definition see appendix 4).

Since we require representation functions to be monotonic we have the theorem

$$subclass(r_1, r_2) \Rightarrow synrep(r_1, p) \sqsubseteq synrep(r_2, p)$$

Since the domain *asyn* of abstract syntax is flat, *proper* is defined for *asyn*. Thus, we have

$$subclass(r_1, r_2) \wedge proper(synrep(r_1, p)) \Rightarrow synrep(r_1, p) = synrep(r_2, p) \quad (*)$$

In other words, extending a reference class r_1 to r_2 will not change proper objects defined in terms of r_1.

The predicates *proper* and *subclass* are easily expressed in the verifier's assertion language (see appendix 4). The documentation of the parser includes

suitable assertions involving *proper* and *subclass*. The fact that objects of abstract syntax remain unchanged after extension operations is immediately deducible by the verifier's prover from this documentation and lemma (*).

4. Static semantics

The static semantics is defined with recursive functions *e*, *t*, etc (see appendix 1). The implementation of these recursive functions is trivial except for one important point. The meaning of recursive type declarations is defined as a least fixed point. This fixed point cannot be computed by simple iteration as this would lead to a nonterminating computation. Therefore, we use the concept of *operationalization* of fixed points introduced in chapter II.

In this section we discuss the application of operationalization to the definition of types in *LS*. Further we outline the development cycle of a semantic analysis program is some detail.

4.1. Recursive declarations

In the definition of *d* we have the clause

$$d[\![type\Delta_{t1}; \ldots; \Delta_{tn}]\!]\varsigma = fix(dt^*[\![\Delta_{t1}; \ldots; \Delta_{tn}]\!]\varsigma)$$

which describes the effect of a type declaration on the environment. In order to determine the validity of programs we have to be able to compute a representation of the environment given by this fixed point. Note, that it is important here, that we compute the *least* fixed point. For example, a non-minimal fixed point could be one in which arbitrary new identifiers are defined in the environment. Consequently, weak axiomatization is insufficient.

4.1.1. Operationalization

The solution to the above problem is operationalization of fixed points as introduced in chapter II. The situation here is much more complicated than in the trivial example presented in chapter II. Therefore we will give a slightly simplified description of our solution.

The reader might find the concept of operationalization very unnatural but it has a very intuitive interpretation in our particular case; most compiler constructors would proceed in the very same way. We define the domain R of unresolved references as:

$$u \in R = (Ty \times Id)^*.$$

An element in R is a list of pairs (r, I) of a type r and an identifier I.

The meaning of such a pair is that the type r is not yet completely defined. Rather, I has been used as a type identifier but has not been previously declared; r is the (as yet unknown) type denoted by I. This may seem contradictory; how can someting unknown be included in a list?. The solution is that we store types in a reference class. An "unknown" type is a pointer to an as yet uninitiallized element of the reference class. It is this pointer that is included in the list of unresolved references.

Once all type declarations are elaborated, all identifiers in the list of unresolved references are defined (if not, this indicates a semantics error). At this point the unresolved references can be "resolved" by replacing incomplete types by the types denoted by the corresponding identifiers. More precisely, we can update the cells pointed to by pointers in the list of unresolved references.

The key idea of operationalization of a fixed point $fix\ f$ where $f \in U_s \to U_s$ is

- to find a representation of the function domain $U_s \to U_s$,
- to compute the representation of f, and
- to provide an operation on the representation of $U_s \to U_s$ which computes the representation of the least fixed point.

In our case we have $dt^* \in Tdef^* \to U_s \to U_s \to U_s$ and want to compute $fix\ dt^*[\![\ldots]\!]\varsigma$, i.e. the argument to fix is in $U_s \to U_s$. Suppose we had a representation, say domain D, for objects in $U_s \to U_s$. Then we want to compute $dt^* \in Tdef^* \to U_s \to D$ and have an operation on D which computes a representation of the least fixed point.

We will use pairs $(\varsigma, u) \in U_s \times R$ to represent elements in $U_s \to U_s$. A pair (ς, u) represents a function, which if applied to ς_1 returns a modified version of ς in which all unresolved references in u are resolved by looking up yet undeclared identifiers in ς_1. To understand the resolution process we have to define the representations used for types, modes, and environments.

A type is represented by a pair consisting of a pointer t_p and a reference class $\#t$. Thus, we have $typerep \in \#T \to T_p \diagdown\!\!\!\diagup Ty$. At this point we are merely interested in functionalities, a precise definition will be given later. Similarly, a mode is represented by a pointer m_p and a reference class $\#m$. But since a component of a mode can be a type a mode also depends of the reference class $\#t$; we have $moderep \in \#M \to \#T \to M_p \diagdown\!\!\!\diagup Md$.

Finally, environments are represented by records and pointers. Since environments contain modes as well as objects from the abstract syntax the

representation is given by

$$envrep \in \#A \to \#M \to \#T \to \#E \diagdown\!\!\!\diagup U_s.$$

If during the evaluation of t_f we encounter an undefined type identifier, a new cell in $\#t$ is created but not further initialized. It is only relevant to have a new unique pointer which together with the undeclared identifier is entered in the list of unresolved references. We call those uninitialized types *anonymous*.

Given this setup resolve is very simple. Suppose we have an element f in $U_s \to U_s$ represented by the pair (ς, u) where $\varsigma = envrep(\#a, \#m, \#t, \#e, z)$. To determine a representation for $fix\ f$ we simply consider all pairs (p, I) in u; for each pair we determine the type denoted by I in ς and update the cell $\#t \subset p \supset$ accordingly. The analogy to the example presented in chapter II should be obvious.

4.1.2. Revised definition of t and dt

We now define revised versions of some valuations which instead of using a second environment to look up forward references build a list of unresolved references whenever they encounter a forward reference during the evaluation. The following arguments are not strictly formal though. To be precise we would have to define the functions below in terms of the representation of types, modes, and environments rather than abstract types, modes, and environments. We trade formality against simplicity; a formal treatment should be obvious though tedious.

$$\boxed{t_f \in Typ \to U_s \to R \to (Ty \times U_s \times R)}$$

$$\begin{aligned}
t_f[\![I]\!]\varsigma_0\, u &= \textbf{if } \lfloor type\ \tau \rfloor :: \varsigma_0[\![I]\!] \textbf{ then } (\tau, \varsigma_0, u)\\
t_f[\![(I_1, \ldots, I_n)]\!]\varsigma_0\, u &= ([\nu{:}sub{:}1{:}n], \varsigma_2[\mu_i/I_i], u)\\
&\quad \textbf{where } \mu_i = \lfloor const{:}i{:}[\nu{:}i{:}i] \rfloor\\
&\quad \textbf{where } (\nu, \varsigma_2) = newtag\ \varsigma_0\\
t_f[\![E_1 \,..\, E_2]\!]\varsigma_0\, u &= \textbf{if } \lfloor const{:}\epsilon_1{:}\tau_1 \rfloor :: e[\![E_1]\!]\varsigma_0 \textbf{ then}\\
&\quad \textbf{if } \lfloor const{:}\epsilon_2{:}\tau_2 \rfloor :: e[\![E_2]\!]\varsigma_0 \textbf{ then}\\
&\quad (\langle union\ \tau_1\ \tau_2 \rangle, \varsigma_0, u)\\
t_f[\![array[T_1]\, of\ T_2]\!]\varsigma_0\, u &= \textbf{let } (\tau_1, \varsigma_2, u_2) = t_f[\![T_1]\!]\varsigma_0\, u,\\
&\quad (\tau_2, \varsigma_3, u_3) = t_f[\![T_2]\!]\varsigma_2\, u_2 \textbf{ in}\\
&\quad \textbf{if } isindex\ \tau_1 \textbf{ then } (\lceil \nu{:}array{:}\tau_1{:}\tau_2 \rceil, \varsigma_4, u_3)\\
&\quad \textbf{where } (\nu, \varsigma_4) = newtag\ \varsigma_3\\
t_f[\![record\ I_1{:}T_1; I_2{:}T_2; \ldots; I_n{:}T_n\ end]\!]\varsigma_0\, u_0 &=\\
&\quad \textbf{if } distinct\langle I_1, \ldots, I_n \rangle \textbf{ then}\\
&\quad \textbf{let } (\tau_i, \varsigma_i, u_i) = t_f[\![T_i]\!]\varsigma_{i-1}\, u_i \textbf{ in}\\
&\quad \textbf{if } (\nu, \varsigma_{n+1}) = newtag\ \varsigma_n \textbf{ then}\\
&\quad (\lceil \nu{:}record{:}\langle I_1{:}\tau_1, \ldots, I_n{:}\tau_n \rangle \rceil, \varsigma_{n+1}, u_n)\\
t_f[\![\uparrow I]\!]\varsigma_0\, u &= \quad (\lceil \nu{:} \uparrow {:}\tau \rceil, \varsigma_1, u_1)\\
&\quad \textbf{where } (\nu, \varsigma_1) = newtag\ \varsigma_0\\
&\quad \textbf{where } (u_1, \tau) = anonymous[\![I]\!]\ u
\end{aligned}$$

Here, *anonymous* creates a new anonymous type which together with the yet undeclared identifier is added to u. Note, that pointer types are the only case where we allow for forward references to appear; all other types have to be defined previously.

Here is one point where our argument has to be slightly informal because we choose not to present all details of our treatment of recursive declarations. We have to assume that all types τ which are returned by *anonymous* are unique and not equal to any existing type. This is apparent if we imagine that in a detailed treatment *anonymous* will not return a type but rather a *new* pointer to $\#t$.

Other valuations are changed accordingly:

$$\boxed{dt_f \in Tdef \to U_s \to R \to (U_s \times R)}$$

$$dt_f[\![I = T]\!]\varsigma_0\, u = \textbf{let } (\tau, \varsigma_2, u_2) = t_f[\![T]\!]\varsigma_0\, u \textbf{ in } (\varsigma_2[\lfloor type{:}\tau \rfloor / I], u_2)$$

$$\boxed{dt_f^* \in Tdef^* \to U_s \to R \to (U_s \times R)}$$

$$\begin{aligned}
dt_f^*[\![]\!]\varsigma_0\, u &= \varsigma_0\\
dt_f^*[\![\Delta_{t0}; \Delta_{t1}; \ldots; \Delta_{tn}]\!]\varsigma_0\, u &= dt_f^*[\![\Delta_{t1}; \ldots; \Delta_{tn}]\!]\varsigma_1\, u_1\\
&\quad \textbf{where } (\varsigma_1, u_1) = dt_f[\![\Delta_{t0}]\!]\varsigma_0\, u
\end{aligned}$$

4.1.3. Representation of $U_s \to U_s$

We mentioned that a pair $\langle \varsigma, u \rangle$ represents an element in $U_s \to U_s$. Let us now define the necessary representation functions. Not only does a pair $\langle \varsigma, u \rangle$ represent an element in $U_s \to U_s$, we also have that a pair $\langle \tau, u \rangle$ represents an element in $U_s \to Ty$ and $\langle \mu, u \rangle$ represents an element in $U_s \to Md$. Let us first consider types. We define a function ϑ_{Ty} mapping pairs in $Ty \times R$ into $U_s \to Ty$.

$$\boxed{\vartheta_{Ty} \in Ty \to R \to U_s \to Ty}$$

$$\begin{aligned}
&\vartheta_{Ty}\ \tau\ u\ \varsigma =\\
&\quad \textbf{if } u :: \ldots \langle \tau, J \rangle \ldots \textbf{ then } (\textbf{ if } \varsigma[\![J]\!] :: [type{:}\tau_2] \textbf{ then } \tau_2) \textbf{ else}\\
&\quad \textbf{if } \tau :: [\nu{:}sub{:}\iota_1{:}\iota_2] \textbf{ then } \tau \textbf{ else}\\
&\quad \textbf{if } \tau :: [\nu{:} \uparrow \tau_3] \textbf{ then } [\nu{:} \uparrow (\vartheta_{Ty}\ \tau_3\ u\ \varsigma)]\\
&\quad \textbf{if } \tau :: [nil] \textbf{ then } \tau\\
&\quad \textbf{if } \tau :: [\nu{:}array{:}\tau_1{:}\tau_2] \textbf{ then } [\nu{:}array{:}(\vartheta_{Ty}\ \tau_1\ u\ \varsigma){:}(\vartheta_{Ty}\ \tau_2\ u\ \varsigma)]\\
&\quad \textbf{if } \tau :: [\nu{:}record{:}\langle I_i{:}p_1, \ldots, I_n{:}p_n \rangle]\\
&\qquad \textbf{then } [\nu{:}record{:}\langle I_i{:}(\vartheta_{Ty}\ \tau_1\ u\ \varsigma), \ldots, I_n{:}(\vartheta_{Ty}\ \tau_n\ u\ \varsigma) \rangle])
\end{aligned}$$

The intuition behind this definition is simple. If τ is not in the list of unresolved references, the result is τ, regardless of ς. If however τ is an anonymous type (or if any component type of τ is) then the type returned is the type denoted by J in ς, where J is the identifier associated with τ in the list of unresolved references.

The new valuations t_f and ϑ_{Ty} are related to t by

$$\begin{aligned}
(\vartheta_{Ty}\ \tau\ u\ \varsigma_1, \varsigma) &= t[\![T]\!]\varsigma_0\varsigma_1\\
&\quad \textbf{where } (\tau, \varsigma, u) = t_f[\![T]\!]\varsigma_0\ \langle\rangle
\end{aligned}$$

Similarly we define ϑ_{Md}. If a mode contains an unresolved type, then this type is determined in the given environment:

$$\boxed{\vartheta_{Md} \in Md \to R \to (U_s \to Md)}$$

$$\begin{aligned}
\vartheta_{Md}\lfloor var{:}p\rfloor\ u\ \varsigma &= \lfloor var{:}(\vartheta_{Ty}\ p\ u\ \varsigma)\rfloor \\
\vartheta_{Md}\lfloor vap{:}p\rfloor\ u\ \varsigma &= \lfloor vap{:}(\vartheta_{Ty}\ p\ u\ \varsigma)\rfloor \\
\vartheta_{Md}\lfloor val{:}p\rfloor\ u\ \varsigma &= \lfloor val{:}(\vartheta_{Ty}\ p\ u\ \varsigma)\rfloor \\
\vartheta_{Md}\lfloor type{:}p\rfloor\ u\ \varsigma &= \lfloor type{:}(\vartheta_{Ty}\ p\ u\ \varsigma)\rfloor \\
\vartheta_{Md}\lfloor const{:}\epsilon{:}p\rfloor\ u\ \varsigma &= \lfloor const{:}\epsilon{:}(\vartheta_{Ty}\ p\ u\ \varsigma)\rfloor \\
\vartheta_{Md}\lfloor sproc\rfloor\ u\ \varsigma &= \lfloor sproc\rfloor \\
\vartheta_{Md}\lfloor proc{:}\dot{\mu}_1,\ldots,\dot{\mu}_n\rfloor\ u\ \varsigma &= \lfloor proc{:}\dot{\mu}_1,\ldots,\dot{\mu}_n\rfloor \\
\vartheta_{Md}\lfloor sfun\rfloor\ u\ \varsigma &= \lfloor sfun\rfloor \\
\vartheta_{Md}\lfloor afun{:}\dot{\mu}_1,\ldots,\dot{\mu}_n{:}p\rfloor\ u\ \varsigma &= \lfloor afun{:}\dot{\mu}_1,\ldots,\dot{\mu}_n{:}(\vartheta_{Ty}\ p\ u\ \varsigma)\rfloor \\
\vartheta_{Md}\lfloor pfun{:}\dot{\mu}_1,\ldots,\dot{\mu}_n{:}p\rfloor\ u\ \varsigma &= \lfloor pfun{:}\dot{\mu}_1,\ldots,\dot{\mu}_n{:}(\vartheta_{Ty}\ p\ u\ \varsigma)\rfloor
\end{aligned}$$

With the above definitions we can define ϑ_U as

$$\boxed{\vartheta_{U_s} \in U_s \to R \to (U_s \to U_s)}$$

$$\vartheta_{U_s}\ \varsigma\ u = \lambda\varsigma_1.(\lambda I.\vartheta_{Md}(\varsigma^{\#1}[\![I]\!])u\ \varsigma_1, \varsigma^{\#2}, \varsigma^{\#3})$$

For ϑ_{U_s}, dt, and dt_f we have the relationship:

$$\begin{aligned}
\vartheta_U\ \varsigma\ u\ \varsigma_1 &= dt[\![I = T]\!]\varsigma_0\varsigma_1 \\
&\quad \text{where } (\varsigma, u) = dt_f[\![I = T]\!]\varsigma_0\ () \\
\\
\vartheta_U\ \varsigma\ u\ \varsigma_1 &= dt^*[\![\Delta]\!]\varsigma_0\varsigma_1 \\
&\quad \text{where } (\varsigma, u) = (dt_f^*[\![\Delta]\!]\varsigma_0\ ()
\end{aligned}$$

4.1.4. Resolving undefined references

Given ς and u, a function *resolve* can be defined, such that

$$resolve\ \varsigma\ u = fix(\vartheta_U\varsigma\ u)$$

resolve will compute a representation of the least fixed point by constructing cyclic lists as outlined in chapter II. Clearly, resolve cannot be defined in terms of abstract entities like environments, modes, and types alone; we have to

define resolve in terms of a particular representation.

$$\begin{aligned}
resolve\ \varsigma\ u = \textbf{if } &(typerep(\#t, p), J)\cdot\dot{u}::u \textbf{ then} \\
&\textbf{let } \dot{\varsigma} = resolve\ \varsigma\ \dot{u} \textbf{ in} \\
&\textbf{let } envrep(\#a, \#m, \#t, \#e, z)::\dot{\varsigma} \textbf{ in} \\
&\textbf{if } \dot{\varsigma}[\![J]\!]::\lfloor type{:}typerep(\#t, q)\rfloor \textbf{ then} \\
&\quad envrep(\#a, \#m, < \#t, \subset p \supset, \#t \subset q \supset >, \#e, z)
\end{aligned}$$

The proof that *resolve* computes a representation of the least fixed point is analogous to the proof of the general theorem given in chapter II and is not repeated here. Note, that *resolve* can be specified by a weak axiomatization; we will add the definition of resolve as well as the definition of t_f and dt_f to the locigal basis of the proof of the static semantics. An implementation and its correctness is then immediate.

4.2. Development of the program

Given the operationalization of fixed points and the according definitions of t_f and dt_f the implementation poses no further problems. In this section we will describe the development process of a part of the static semantics in some detail emphasizing the systematic way in which a program and its specifications can be derived from specifications.

4.2.1. Computing recursive functions

Let us consider the implementation of a procedure that checks expressions for their semantic validity. One clause of the recursive definition of e is:

$$\begin{aligned}
e[\![E_0[E_1]]\!]\varsigma = \textbf{if } &\lfloor \nu{:}array\ r_1\ of\ r_2\rfloor::type\ e[\![E_0]\!]\varsigma \textbf{ then} \\
&\textbf{if } compatible\ r_1(type\ e[\![E_1]\!]\varsigma) \textbf{ then} \\
&\quad \textbf{if } isvar\ e[\![E_0]\!]\varsigma \textbf{ then } \lfloor var{:}r_2\rfloor \textbf{ else} \\
&\quad\quad \textbf{if } isval\ e[\![E_0]\!]\varsigma \textbf{ then } \lfloor val{:}r_2\rfloor
\end{aligned}$$

As a first step we rewrite this definition in an equivalent first order form which can then be input to the verifier. This translation is straightforward. The only point that requires attention is the type of functions involved. For example *compatible* will not return a boolean result, rather an element in $\{TT, FF\}_\perp$. Therefore, truthvalued functions cannot be taken as predicates in the axiomatization. The corresponding rules in the verifier's language are given in figure 5.

Based on this definition we decide to implement e as a recursive procedure Ce in our program. The result will be returned in a var parameter *result*.

```
rulefile(e)
 constant TT;

 replace e(mkeindex(E0, E1), zeta) where
         mkearraytype(nu, tau1, tau2) = typee(e(E0, zeta)) ∧
         compatible(tau1, typee(e(E1, zeta))) = TT ∧
         isvar(e(E0, zeta)) = TT by
                 mkevalmode(tau2);

 replace e(mkeindex(E0, E1), zeta) where
         mkearraytype(nu, tau1, tau2) = typee(e(E0, zeta)) ∧
         compatible(tau1, typee(e(E1, zeta))) = TT ∧
         isval(e(E0, zeta)) = TT by
                 mkevalmode(tau2);
```

Fig. 5

For the first draft we assume that we have functions *iszzz* implementing tests on the abstract syntax as defined earlier. In addition we assume that we have procedures for the decomposition of abstract syntax:

```
function isindex(Ez:Exp):TTFF;
entry true; exit true; external ;

procedure matchindex(Ez:Exp; var E0, E1:Exp);
entry isindex(Ez) = TT; exit Ez = mkeindex(E0, E1); external ;
```

The true entry and exit specifictions cause the verifier to treat *isindex* as a free function symbol. Its meaning is not defined by entry and exit conditions rather it is given as part of the logical basis.

One additional problem is to get the verifier to accept an "abstract" program. In particular this program has to satisfy all declaration and type constraints of Pascal. A simple solution is to define abstract objects as arbitrary enumeration types. We can still take advantage of the system's typechecking but we do not have to define the structure of the data in more detail.

The complete initial program computing *e* (for the case of an indexed expression) is

```
pascal

type Exp= (z1, z2, z3);
     Us= (z4, z5);
```



```
     Md= (z6, z7);
    Typ= (z8, z9);
   TTFF= (TT, FF, UU);

procedure error;
entry true; exit false; external ;

function isindex(Ex: Exp): TTFF;
entry true; exit true; external ;

procedure matchindex(Ex: Exp; var E0, E1: Exp);
entry isindex(Ex) = TT; exit Ex = mkeindex(E0, E1); external ;

function isarraytype(Ty: Typ): TTFF;
entry true; exit true; external ;

procedure matcharraytype(Ty: Typ; var nu: integer; var T0, T1: Typ);
entry isarraytype(Ty) = TT; exit Ty = mkearraytype(nu, T0, T1); external ;

function compatible(tau1, tau2: typ): TTFF;
entry true; exit true; external ;

function isvar(mu: Md): TTFF;
entry true; exit true; external ;

function isval(mu: Md): TTFF;
entry true; exit true; external ;

function mkevalmode(tau: typ): Md;
entry true; exit true; external ;

function typee(mu: Md): Typ;
entry true; exit true; external ;

procedure Ce(Ex: exp; zeta: Us; var result: Md);
entry true; exit e(Ex, zeta) = result;
var E0, E1      : Exp;
  tau, tau1, tau2: typ;
       mu, mu1: Md;
            nu: integer;
begin
...
```

```
  if isindex(Ex) = TT then
  begin
    matchindex(Ex, E0, E1);
    Ce(E0, zeta, mu);
    tau ← typee(mu);
    if not(isarraytype(tau) = TT) then error else
    begin
      matcharraytype(tau, nu, tau1, tau2);
      Ce(E1, zeta, mu1);
      if not(compatible(tau1, typee(mu1)) = TT) then error else
      if (isvar(mu) = TT) or (isval(mu) = TT)
        then result ← mkevalmode(tau2) else error
    end
  end else
  if ...
    ...
  else error
end ;.
```

For this program the verifier produces a set of verification condition. All but one are trivially true since all paths but one contain a call to error. The interesting path is characterized by:

```
(isindex(ex) = tt
=>
  isindex(ex) = tt ∧
  (ex = mkeindex(e0_2, e1_2) ∧
    e(e0_2, zeta) = mu_2 ∧
    ¬¬(isarraytype(typee(mu_2)) = tt)
    =>
      isarraytype(typee(mu_2)) = tt ∧
      (typee(mu_2) = mkearraytype(nu_1, tau1_1, tau2_1) ∧
      e(e1_2, zeta) = mu1_1 ∧
      ¬¬(compatible(tau1_1, typee(mu1_1)) = tt) ∧
      (isvar(mu_2) = tt ∨
      isval(mu_2) = tt)
      =>
        e(ex, zeta) = mkevalmode(tau2_1))))
```

where variables of the form x_1 are introduced by the verifier. It can easily be seen that this verification condition is provable from our axiomatization of e and the verifier's prover handles this case in fractions of a second.

4.2.2. Refinement

The above verification shows us that the overall design of the program to compute e is correct. In order to make Ce an executable procedure we have to define the representation of the domains Exp, Us, Md, Typ, and T. Given a particular representation we can then proceed to refine functions and procedures using these domains.

Let us do this refinement one domain at a time, starting with $TTFF$. We let $truthrep \in boolean \nwarrow\!\!\!\!\searrow TTFF$ with

$$truthrep(true) = TT$$
$$truthrep(false) = FF$$

We now refine the program is such a way that the proof we have already given for the abstract version remains valid. The declaration

```
function isindex(Ex:Exp):TTFF;
entry true; exit true; external;
```

is replaced by the concrete version

```
function Cisindex(Ex:Exp):boolean;
entry true; exit truthrep(Cisindex) = isindex(Ex); external;
```

and the call

```
if isindex(Ex) = TT then ...
```

becomes

```
if Cisindex(Ex) then ...
```

We do these changes systematically for all occurrences of $TTFF$. The resulting program gives rise to the verification condition given in figure 6.

Applying the definition of *truthrep* this verification condition can be transformed into a formula equivalent to the verification condition for the initial program, see figure 7. Consequently, if the refinement is done systematically as outlined above, the verification conditions for the refined version of the program are provable with the same logical basis as the original program. The only additional facts required are the definitions of the representation functions.

4.2.3. Representation

Similar to the implementation of $TTFF$ we define representations for the abstract domains still undefined and refine the program in the manner outlined above. We will now discuss the representations used for the various domains but will not go through any more refinement steps.

```
(truthrep(cisindex(ex)) = isindex(ex) ∧
cisindex(ex)
=>
  isindex(ex) = tt ∧
  (ex = mkeindex(e0_2, e1_2) ∧
    e(e0_2, zeta) = mu_2 ∧
    truthrep(cisarraytype(typee(mu_2))) = isarraytype(typee(mu_2)) ∧
    ¬¬cisarraytype(typee(mu_2))
    =>
      isarraytype(typee(mu_2)) = tt ∧
      (typee(mu_2) = mkearraytype(nu_1, tau1_1, tau2_1) ∧
      e(e1_2, zeta) = mu1_1 ∧
      truthrep(ccompatible(tau1_1, typee(mu1_1))) =
                    compatible(tau1_1, typee(mu1_1)) ∧
      ¬¬ccompatible(tau1_1, typee(mu1_1)) ∧
      truthrep(cisvar(mu_2)) = isvar(mu_2) ∧
      truthrep(cisval(mu_2)) = isval(mu_2) ∧
      (cisvar(mu_2) ∨
      cisval(mu_2))
      =>
        e(ex, zeta) = mkevalmode(tau2_1))))
```

Fig. 6

Obviously, the representation of the abstract syntax has to be identical to that used in the parser. A conversion to a different representation is not warrented since the existing one suffices.

Types are represented by the following record structure:

```
ttype =↑ tnode;
tnode =  record
            tkind:(tagrecordtype, tagarraytype,
                   tagsubtype, tagpointertype,
                   tagniltype);
            typetag:integer;
            lwb:integer;
            upb:integer;
            subt1:ttype;
            subt2:ttype;
            recs:ttype;
            id:integer
         end;
```

The corresponding representation function *typerep* is defined as


```
(tt = isindex(ex) ∧
=>
  isindex(ex) = tt ∧
  (ex = mkeindex(e0_2, e1_2) ∧
    e(e0_2, zeta) = mu_2 ∧
    tt = isarraytype(typee(mu_2))
    =>
      isarraytype(typee(mu_2)) = tt ∧
      (typee(mu_2) = mkearraytype(nu_1, tau1_1, tau2_1) ∧
      e(e1_2, zeta) = mu1_1 ∧
      tt = compatible(tau1_1, typee(mu1_1)) ∧
      (tt = isvar(mu_2) ∧
        truthrep(cisval(mu_2)) = isval(mu_2)
        ∨
        truthrep(cisvar(mu_2)) = isvar(mu_2) ∧
        tt = isval(mu_2))
      =>
        e(ex, zeta) = mkevalmode(tau2_1))))
```

Fig. 7

```
typerep(#t, p) = if p = nil then ⊥ else
    let r = #t ⊂ p ⊃ in
      if t.mkind = tagrecordtype then
        [ν:record:rs] else
      if t.mkind = tagarraytype then
        [ν:array:r1:r2] else
      if t.mkind = tagsubtype then
        [ν:sub:ι1:ι2] else
      if t.mkind = tagpointertype then
        [ν:↑ r1] else
        [nil]
    where ν = tagrep(r.typetag)
    where rs = recrep(#t, r.recs)
    where r1 = typerep(#t, r.subt1)
    where r2 = typerep(#t, r.subt2)
    where ι1 = intrep(#t, r.lwb)
    where ι2 = intrep(#t, r.upb)
```

Modes are represented in a similar way by:

```
mode =↑ mnode;
mnode =  record
```

```
        mkind:(tagvalmode, tagvapmode, tagafuncmode, tagsfuncmode,
                tagvarmode, tagtypemode, tagprocmode, tagsprocmode,
                tagpfuncmode, tagconstmode);
        ty:ttype;
        mlist:mode;
        next:mode;
        val:integer
    end;
```

For environments we use a simple minded implementation consisting of two linked lists and an integer number. The two lists represent the functions $Id \rightarrow Md$ and $Num \rightarrow T$ respectively (similar to association lists in Lisp). The number component gives the highest type tag used so far. Since we use type tags consecutively starting with $int = 0$, $bool = 1$ this implicitely gives us a mapping from type tags into T.

A more sophisticated implementation can be substituted easily without invalidating other parts of the proof.

4.2.4. Auxilliary functions

For abstract syntax, types and modes we provide tests, constructors, and matching functions. Typical examples are:

```
function Cisvalmode(mu:mode):boolean;
global  (#tnode, #mnode);
  entry true;
    exit isvalmode(moderep(#mnode, #tnode, mu)) = truthrep(Cisvalmode);
begin
  Cisvalmode ← mu ↑ .mkind = tagvalmode;
end ;

procedure Cmkeprocmode(mul:mode; var mu:mode);
global  (#enode, #mnode, #tnode, #anode);
  entry true;
    exit moderep(#mnode, #tnode, mu) =
          mkeprocmode(moderep(#mnode, #tnode, mul));
begin
  new(mu);
  mu ↑ .mkind ← tagprocmode;
  mu ↑ .mlist ← mul;
end ;

procedure matchvalmode(mu:mode; var tau:ttype);
```



```
global  (#mnode, #tnode);
  entry isvalmode(moderep(#mnode, #tnode, mu)) = TT;
    exit moderep(#mnode, #tnode, mu) = mkevalmode(typerep(#tnode, tau));
begin
  tau ← mu ↑ .ty;
end ;
```

A function error is provided and serves the same purpose as in the parser. The false exit condition guarantees that all paths are verifiable if they contain a call to *error*.

No implementation is provided for o, w, sf, and sp. An implementation is immediate, given a concrete set of operators and special procedures and functions

Other auxilliary functions are implemented following their recursive definition. An example is

```
function Cisreturnable(Ty:ttype):boolean;
global (#enode, #mnode, #tnode, #anode);
  exit truthrep(Cisreturnable) = isreturnable(typerep(#tnode, Ty));
begin
  Cisreturnable ← Cissubtype(Ty) or
      Cispointertype(Ty) or Cisniltype(Ty);
end ;
```

4.2.5. The complete program

We have described the development of Ce, the implementation of e, in some detail. Also, the theory to compute recursive type declarations as been presented. The implementation of the remaining functions is straightforward. In apendix 4 we include several examples, in particular the part of the program dealing with recursive type declarations.

5. Code generation

In all previous parts of the compiler we had to prove that a program computes a particular function. In the code generation the situation is different. The code to be produced does not depend functionally on the input program. We have considerable freedom as to what kind of code we produce subject only to the restriction that it has the same semantics as the input program.

One possible approach to code generation is to prove that some abstract "code generating function" produces correct code. This function can then be

taken as specification for a concrete code generator. For example, Milne and Strachey [MS76] provide a suitable code generating function for the language SAL. They give manual proofs of the correctness of this code generation function. Similar proofs are checked mechanically with LCF by A. Cohn [Co79b].

We could have chosen to use a code generating function as specification for our code generation. The techniques for implementation would have been similar to those used in other parts of the compiler. We choose not to do so. Rather, we consider a detailed enough semantics of the source language and an abstract enough definition of the target language such that equivalence between source and target programs can be expressed in the verifier's assertion language.

Still, we need fairly elaborate theorems about source and target semantics and their relation. We omit most of the formal proofs of the necessary semantic theories as these are fairly well understood [MS76]. Rather we concentrate on issues related to program verification; how can the correctness of our code generator be specified? what are the general principles of proof employed?.

5.1. Principle of code generation

The code generating program is executed after the semantics analysis. Thus, we may assume that the input is a semantically valid abstract syntax tree. The code generation is performed by a set of recursive procedures corresponding to meaning functions of the semantic definition of *LS*. These functions recursively work their way down the abstract syntax tree and generate the appropriate code. For example let us consider expressions.

The definition of $\hat{\mathcal{E}}$ contains the following definitional clause for binary operators:

$$\hat{\mathcal{E}}[\![E_0\ \Omega\ E_1]\!]\varsigma\rho\gamma = \mathcal{R}[\![E_0]\!]\varsigma\rho;\ \mathcal{R}[\![E_1]\!]\varsigma\rho;\ binop[\![\Omega]\!]\gamma$$

Corresponding to $\hat{\mathcal{E}}$ and $\mathcal{R}$ we have a procedure *AEcode* and *Rcode*. In the case of binary operators *AEcode* proceeds as follows:

- Call *Rcode* for the first expression. The resulting code will guarantee that after its execution the value of the first expression is on top of the stack.
- Call *Rcode* for the second expression; append the new code to the previously generated code. Now, at this point during runtime the results of both expressions will be the two topmost elements of the stack.
- Generate code to execute the binary operation in question. This code will remove the two topmost elements and push the result of the operation.

Thus altogether the call to *Ecode* generates code that leaves the result of the expression $E_0\ \Omega\ E_1$ on top of the stack.

Rcode itself is similar to *AEcode* but merely adds possible coercions of L-values (*Lv*) to values (*V*).

One immediate question is what happens for an expression that is just a variable? Well, we have to generate code that accesses the location associated with this variable. But since the store in the source and target language are different we have to introduce a "compile time environment" to record the necessary information about locations assigned to variables in the target language we introduce a "compile time environment". A compile time environment is similar to an environment except that it maps identifiers into locations in the target semantics rather than into locations in the source semantics.

If the code generating procedures encounter a declaration new locations on the target machine are allocated. The corresponding identifiers are bound to these new locations in the compile time environment.

By the choice of the definitional methods for source and target languages addressing is relative in *LS* and *LT*. In the produced code the relative addresses are hard-wired while the base addresses are only known at run time; the final address computation determining the absolute address is done at execution time.

Special considerations are necessary to treat labels properly. Consider a conditional statement **if** E **then** Γ_0 **else** Γ_1 for which we generate the following code:

- generate two new unique labels N_0, and N_1.
- generate code to evaluate E
- generate a conditional jump to N_0 if E is false
- generate code for Γ_0
- generate an unconditional jump to N_1
- define label N_0 in the output code
- generate code for Γ_1
- define label N_1 in the output code

This code sequence introduces new labels in the target program which have no counter part in the source language. Recall that the definition of *LT* considers all labels of a target program at the same time (outer level); i.e. one fixed point determines all label continuations. The equivalence of a target program with a source program is therefore very hard to establish if there is

not a one to one correspondence of labels.

We will solve this problem in two steps. First, we change the definition of the semantics of conditionals in the source language such that it uses explicit labels and jumps. We prove that this new definition is equivalent to the original one.

In addition we introduce a block construct in the target language that allows us to encapsulate labels. A block $(I_1, \ldots, I_n)$ is considered a single instruction; internally a block consists of a list of instructions, possibly other blocks. All labels defined inside a block are local to this block. Thus, for the above example we will generate one block of code. Auxiliary labels are hidden inside the block; furthermore we have a one to one correspondence of statements of LS and instructions (blocks) of LT.

5.2. Modified semantics definitions

In this section we define a block construct as an extension to the target language LT. We show how the extended language relates to the original language. We prove a theorem stating that under certain weak conditions programs in the extended language can be transformed into programs in LT.

Furthermore, we present a revised definition for some statements of LS in terms of conditionals and explicit jumps. We prove the equivalence with the original definition.

5.2.1. A structured target language

To solve the label problem mentioned above we will introduce a block structure in the target language in such a way that label values within a block can be determined independently from the rest of the program. This will allow to translate, say a while loop, into some code block whose correctness we can determine without looking at the context.

We will prove a theorem stating that under very weak assumptions a block of code can be textually expanded (block boundaries can be omitted) without altering the meaning of the program. In generating code in the compiler we will prove that the necessary conditions are satisfied. Therefore the generated block structured code will be equivalent to the same code with block boundaries ignored.

Let us define a different language LT2 by adding the instruction $(I_1, \ldots, I_n)$ to the instruction set of LT with the semantics

$$\mathcal{M}[\![(I_1, \ldots, I_n)]\!]\rho\gamma = \textbf{ if } \mathit{distinct}(l[\![I_1 \ldots I_n]\!]) \textbf{ then } \mathcal{M}^*[\![I_1, \ldots, I_n]\!]\rho_1\gamma$$

where $\rho_1 = fix(\lambda\hat{\rho}.\rho[\mathcal{L}[\![I_1 \ldots I_n]\!]\hat{\rho}\gamma/l[\![I_1 \ldots I_n]\!]])$.

In effect, LT2 allows for programs as instructions. The importance is that labels within each program are only visible locally. We will now show that a program in $p \in LT2$ is equivalent to p with all parentheses omitted, provided all labels in the resulting program are distinct.

Let S_i be sequences of statements and p, q, and r programs such that

$$p = S_1\, q\, S_3\,, \quad q = (S_2), \text{ and } r = S_1\, S_2\, S_3$$

We call r the "flat" version of p, i.e. in the flat version block boundaries are omitted. Let us introduce the abbreviations $\mathcal{L}_i = \mathcal{L}[\![S_i]\!]$, $l_i = l[\![S_i]\!]$, and $\mathcal{M}_i = \mathcal{M}[\![S_i]\!]$.

The condition under which a program is equivalent to its flat version is that labels defined in S_1 and S_3 are distinct from those defined in S_2 and that there are no jumps into inner blocks. Clearly, if a LT program is well formed, jumps into inner blocks are impossible since inner labels are not visible.

We write $\rho =_2 \overline{\rho}$ for two environments that only differ in the values bound to labels in l_2. Formally, $\rho =_2 \overline{\rho}$ holds if $\forall l.l \notin l_2 \rightarrow \rho[\![l]\!] = \overline{\rho}[\![l]\!]$. The LT program p (as above) is well formed if $\rho =_2 \overline{\rho}$ implies that $\mathcal{M}_1\rho = \mathcal{M}_1\overline{\rho}$ and $\mathcal{M}_3\rho = \mathcal{M}_3\overline{\rho}$.

Theorem If $l_1 \cdot l_2 \cdot l_3$ are distinct and if p is well formed then $\mathcal{M}[\![p]\!]\rho\gamma = \mathcal{M}[\![r]\!]\rho\gamma$.

Proof Let ρ_p, ρ_q, and ρ_r be the environments in which the (top level) statements of programs p, q, and r are evaluated. By definition of $\mathcal{M}$ we have

$$\begin{aligned}
\rho_q &= \textbf{fix } \lambda\overline{\rho}.\rho_p[\mathcal{L}_2\overline{\rho}; \mathcal{M}_3\rho_p\gamma/l_2] \\
\rho_p &= \textbf{let } \rho_z = \textbf{fix } \lambda\hat{\rho}.\overline{\rho}[\mathcal{L}_2\hat{\rho}; \mathcal{M}_3\overline{\rho}\gamma/l_2] \textbf{ in} \\
&\quad \textbf{fix } \lambda\overline{\rho}.\rho[\mathcal{L}_1\overline{\rho}; \mathcal{M}_2\rho_z; \mathcal{M}_3\overline{\rho}\gamma/l_1][\mathcal{L}_3\overline{\rho}\gamma/l_3] \\
\rho_r &= \textbf{fix } \lambda\overline{\rho}.\rho[\mathcal{L}_1\overline{\rho}; \mathcal{M}_2\overline{\rho}; \mathcal{M}_3\overline{\rho}\gamma/l_1][\mathcal{L}_2\overline{\rho}; \mathcal{M}_3\overline{\rho}\gamma/l_2][\mathcal{L}_3\overline{\rho}\gamma/l_3]
\end{aligned}$$

First we prove that $\rho_q = \rho_r$; that is, the instruction sequence S_2 is evaluated in the same environment as the flat version r of p. Let us rewrite the fixed points as equivalent recursion equations. We can define ρ_p and ρ_q by a mutually recursive definition as follows

$$\begin{pmatrix} \rho_q \\ \rho_p \\ \rho_z \end{pmatrix} = \begin{pmatrix} \rho_p[\mathcal{L}_2\rho_q; \mathcal{M}_3\rho_p\gamma/l_2] \\ \rho[\mathcal{L}_1\rho_p; \mathcal{M}_2\rho_z; \mathcal{M}_3\rho_p\gamma/l_1][\mathcal{L}_3\rho_p\gamma/l_3] \\ \rho_p[\mathcal{L}_2\rho_z; \mathcal{M}_3\rho_p\gamma/l_2] \end{pmatrix}$$

Similarly we can define ρ_r as

$$\begin{pmatrix}\rho_r\\ \rho_y\end{pmatrix}=\begin{pmatrix}\rho_y[\mathcal{L}_2\rho_r;\mathcal{M}_3\rho_r\gamma/l_2]\\ \rho[\mathcal{L}_1\rho_r;\mathcal{M}_2\rho_r;\mathcal{M}_3\rho_r\gamma/l_1][\mathcal{L}_3\rho_r\gamma/l_3]\end{pmatrix}$$

First, observe, that the recursion equation for ρ_q and ρ_z are identical (up to renaming of the variables). Thus, we can eliminate ρ_z without changing the fixed points for ρ_q and ρ_p.

$$\begin{pmatrix}\rho_q\\ \rho_p\end{pmatrix}=\begin{pmatrix}\rho_p[\mathcal{L}_2\rho_q;\mathcal{M}_3\rho_p\gamma/l_2]\\ \rho[\mathcal{L}_1\rho_p;\mathcal{M}_2\rho_q;\mathcal{M}_3\rho_p\gamma/l_1][\mathcal{L}_3\rho_p\gamma/l_3]\end{pmatrix}$$

Since we have

$$\rho_r=\rho_y[\mathcal{L}_2\rho_r;\mathcal{M}_3\rho_r\gamma/l_2]$$

ρ_r and ρ_y only differ for labels in l_2. Thus, by definition of $=_2$ we have $\rho_r =_2 \rho_y$. By our assumptions of well formedness we have $\mathcal{L}_1\rho_z = \mathcal{L}_1\rho_y$, $\mathcal{L}_3\rho_z = \mathcal{L}_3\rho_y$, and $\mathcal{M}_3\rho_z = \mathcal{M}_3\rho_y$. Therefore, we can rewrite the definition of ρ_r as

$$\begin{pmatrix}\rho_r\\ \rho_y\end{pmatrix}=\begin{pmatrix}\rho_y[\mathcal{L}_2\rho_r;\mathcal{M}_3\rho_r\gamma/l_2]\\ \rho[\mathcal{L}_1\rho_y;\mathcal{M}_2\rho_r;\mathcal{M}_3\rho_y\gamma/l_1][\mathcal{L}_3\rho_y\gamma/l_3]\end{pmatrix}$$

But now ρ_r, ρ_y and ρ_q, ρ_p are defined by isomorphic recursion equations and thus must have equal fixed points. We conclude $\rho_p = \rho_y$ and $\rho_q = \rho_r$ and $\rho_p =_2 \rho_r$.

The theorem follows immediately. Since $\rho_p =_2 \rho_r$ we have $\mathcal{M}[\![S_1]\!]\rho_p = \mathcal{M}[\![S_1]\!]\rho_r$ and $\mathcal{M}[\![S_3]\!]\rho_p = \mathcal{M}[\![S_3]\!]\rho_r$. Furthermore, $\mathcal{M}[\![q]\!]\rho_p = \mathcal{M}[\![S_2]\!]\rho_q = \mathcal{M}[\![S_2]\!]\rho_r$ which completes the proof. ∎

Corollary If the restrictions on the use of labels for the theorem are satisfied, then every program is equivalent to its flat version

Proof Immediate, since any program has only a finite nesting the theorem can be applied to the program repeatedly. ∎

5.2.2. A modified definition of *LS*

We add the statement *unless E goto N* to the abstract syntax of *LS*. We define

$$\mathcal{B}[\![\textit{unless } E \textit{ goto } N]\!]\varsigma\rho = \mathcal{R}[\![E]\!]\varsigma\rho; Cond(\gamma, \rho[\![N]\!])$$

This new statement is not available to the *LS* programmer, i.e. no external syntax is provided. Rather, the unless statement is used to define conditionals and loops as follows:

$$\begin{aligned}\mathcal{B}[\![\textit{if } E \textit{ then } \Gamma_0 \textit{ else } \Gamma_1]\!]\varsigma\rho = \mathcal{C}\ell[\![&\textit{unless } E \textit{ goto } N_0;\\ &\textit{begin } \Gamma_0 \textit{ end};\\ &\textit{goto } N_1;\\ &N_0\textit{:begin } \Gamma_1 \textit{ end};\\ &N_1\textit{:dummy}]\!]\varsigma\rho\gamma\\ \text{where } \varsigma[\![N_1]\!] = \textit{false}, &\varsigma[\![N_0]\!] = \textit{false} \wedge N_0 \neq N_1\end{aligned}$$

$$\begin{aligned}\mathcal{B}[\![\textit{while } E \textit{ do } \Gamma \textit{ od}]\!]\varsigma\rho\gamma = \mathcal{C}\ell[\![&N_0\textit{: unless } E \textit{ goto } N_1;\\ &\textit{begin } \Gamma \textit{ end};\\ &\textit{goto } N_0;\\ &N_1\textit{: dummy}]\!]\varsigma\rho\gamma\\ \text{where } \varsigma[\![N_1]\!] = \textit{false}, &\varsigma[\![N_0]\!] = \textit{false}, N_0 \neq N_1\end{aligned}$$

$$\begin{aligned}\mathcal{B}[\![\textit{repeat } \Gamma \textit{ until } E]\!]\varsigma\rho\gamma = \mathcal{C}\ell[\![&N_1\textit{: begin } \Gamma \textit{ end};\\ &\textit{unless } E \textit{ goto } N_1]\!]\varsigma\rho\gamma\\ \text{where } \varsigma[\![N_1]\!] = \textit{false}&\end{aligned}$$

Theorem The new definitions for conditionals, while and repeat loops are equivalent to those of the original definition (appendix 1).

Proof We consider only while loops. Other cases follow by similar arguments. By the original definition we have

$$\mathcal{B}[\![\textit{while } E \textit{ do } \Gamma \textit{ od}]\!]\varsigma\rho\gamma = \mathbf{fix}\,(\lambda\dot{\gamma}.\mathcal{R}[\![E]\!]\varsigma\rho; cond(C\ell[\![\Gamma]\!]\varsigma\rho\dot{\gamma}, \gamma))$$

It is to show that

$$\begin{aligned}\mathbf{fix}\,(\lambda\dot{\gamma}.\mathcal{R}[\![E]\!]\varsigma\rho; cond(C\ell[\![\Gamma]\!]\varsigma\rho\dot{\gamma}, \gamma)) = \\ \mathcal{C}\ell[\![&N_0\textit{: unless } E \textit{ goto } N_1;\\ &\textit{begin } \Gamma \textit{ end};\\ &\textit{goto } N_0;\\ &N_1\textit{: dummy}]\!]\varsigma\rho\gamma\\ \text{where } \varsigma[\![N_1]\!] = \textit{false}, &\varsigma[\![N_0]\!] = \textit{false}, N_0 \neq N_1\end{aligned}$$

Let us abbreviate N_0: *unless E goto* N_1; *begin* Γ *end*; *goto* N_0; N_1: *dummy* as Γ_{new}. By the definition of $C\ell$ the right hand side of the above equation becomes

$$C[\![\Gamma_{new}]\!]\varsigma\dot{\rho}$$

where $\dot{\varsigma} = \varsigma[(true, true)/\langle N_0, N_1\rangle]$ and $\dot{\rho} = \mathbf{fix}\,(\lambda\bar{\rho}[J[\![\Gamma_{new}]\!]\dot{\varsigma}\bar{\rho}\gamma/\langle N_0, N_1\rangle])$. Evaluating J we get $\dot{\rho} = \mathbf{fix}\,(\lambda\bar{\rho}[C[\![\Gamma_{new}]\!]\dot{\varsigma}\bar{\rho}\gamma/N_0][\gamma/N_1])$.

Now consider $C[\![\Gamma_{new}]\!]\dot{\varsigma}\bar{\rho}\gamma$; after some simplification we get

$$C[\![\Gamma_{new}]\!]\dot{\varsigma}\bar{\rho}\gamma = \mathcal{R}[\![E]\!]\dot{\varsigma}\bar{\rho}; cond(C\ell[\![\Gamma]\!]\dot{\varsigma}\bar{\rho}\gamma, \bar{\rho}[\![N_0]\!]).$$

By the general rule

$$fix(\lambda\dot{\rho}.\rho[f(\dot{\rho}[\![x]\!])/x]) = \rho[\,\mathbf{fix}\,(\lambda\dot{\gamma}.f(\dot{\gamma}))/x]$$

we derive

$$\dot{\rho}[\![N_0]\!] = \mathbf{fix}\,(\lambda\dot{\gamma}.\mathcal{R}[\![E]\!]\dot{\varsigma}\dot{\rho}; cond(C\ell[\![\Gamma]\!]\dot{\varsigma}\dot{\rho}\gamma, \dot{\gamma}))$$

Since $\rho[\![N_i]\!] = false$ the new labels N_0 and N_1 cannot appear as global labels of either E or Γ. Therefore we have

$$C\ell[\![\Gamma]\!]\dot{\varsigma}\dot{\rho}\gamma = C\ell[\![\Gamma]\!]\varsigma\rho\gamma$$

and

$$\mathcal{R}[\![E]\!]\dot{\varsigma}\dot{\rho}\gamma = \mathcal{R}[\![E]\!]\varsigma\rho\gamma$$

which proves the theorem. ∎

5.3. Relation between *LS* and *LT*

Asserting that a *LS* and a *LT* program have the same meaning is easy since their respective domains of meanings are identical ($N^* \to N^* + E_R$). This situation is more complicated at intermediate stages of a computation.

Consider again the example of expressions given earlier. Not only do we have to prove that the correct result is pushed on the stack by the generated code; we also have to prove that evaluating an expression causes "corresponding" side effects. In the following discussion we will make the notion of "corresponding meaning" precise.

We introduce a set of predicates that relate objects in the source language to corresponding objects in the target language. For example, answers (A) in *LS* are related to answers in *LT* by equality, i.e. answers correspond if and only if they are equal. We will use indices S and T to disambiguate objects of source and target semantics that are denoted by the same symbol. We define

$$\vartheta_A(a_S, a_T) \equiv a_S = a_T$$

For other domains D the relations ϑ_D are not as trivial.

The central question is how objects of *LS* are stored in the target language and how the relation between locations of *LS* and *LT* can be expressed. Also, it has to be clarified how other declared objects such as procedures, functions and labels are implemented in *LT*.

We use "compile time environments" to relate labels and procedure identifiers of *LS* to labels of *LT* and variable identifiers to relative addresses of *LT*.

"Relative storage maps" relate relative locations of *LS* to relative locations of *LT*.

Finally, "absolute storage maps" relate absolute locations of *LS* and *LT*.

5.3.1. Compile time environments

Given a variable, procedure, or function identifier or a numeral denoting a label in *LS*. A compile time environment $y \in Y$ specifies what the corresponding objects in *LT* are. We define Y as

$$y \in Y = (Id \to N) \times (Id \to N) \times (Num \to N) \times N$$

The first component is a mapping from identifiers to relative locations; the second component maps procedure and function identifiers into labels. The component $Num \to N$ maps labels of *LS* into labels of *LT*. The last component N is mainly used for bookkeeping purposes; it specifies the number of memory cells that are allocated in the current lexical level.

5.3.2. Storage allocation

Let us now decide how data objects of *LS* are to be stored in *LT*. All data objects other than arrays and records are stored in one cell. All elements of an array have the same size (require the same number of cells); furthermore the element size is known at compile time. Therefore, all elements of an array are stored in consecutive locations. The first of these locations is the address of the array. The situation is similar for records. Since number and size of the components are known at compile time all elements are stored consecutively. Again, the first address is the address of the record.

To access a component of a record or an array we need the address (first element) of the record or array. Given an index or a selector the offset from the first element address can be computed.

Let *size* be a function determining the number of cells required to store a variable of type τ.

$$\boxed{size \in Ty \to N}$$

$$\begin{aligned}
size[\nu{:}sub{:}\iota_1{:}\iota_2] &= 1\\
size[\nu{:}\uparrow\tau] &= 1\\
size[nil] &= 1\\
size[\nu{:}array{:}\tau_1{:}\tau_2] &= \textbf{if } \tau_1{::}[\dot\nu{:}sub{:}\iota_1{:}\iota_2] \textbf{ then}\\
&\qquad (size\ \tau_2)^*(\iota_2 - \iota_1 + 1)\\
size[\nu{:}record{:}(I_1{:}\tau_1, \ldots, I_n{:}\tau_n)] &= (size\ \tau_1) + \ldots + (size\ \tau_n)
\end{aligned}$$

Given an object of type τ with the L–value a which is assigned the address z in the target language we can define how addresses of components of this object are related to target addresses. We define a function $ad \in Lv_s \to Ty \to N \to ((L_s \times N) \to T)$ which given a L–value, a type and an address in LT returns a relation $\in (L_s \times N) \to T^2$ between locations of LS and addresses in LT.

$$\begin{aligned}
ad\ ar\ z = \textbf{if } a \in L_s \textbf{ then } &\{a, z\} \textbf{ else}\\
&\textbf{if } \tau{::}[\nu{:}array{:}[\dot\nu{:}sub{:}\iota_1{:}\iota_2]{:}\tau_2] \textbf{ then}\\
&\quad ad(a\iota_1)\tau_2\ z \bigcup\\
&\quad ad(a(\iota_1 + 1))\tau_2(z + size\ \tau_2) \bigcup\\
&\quad \ldots\\
&\quad ad(a\iota_2)\tau_2(z + (\iota_2 - \iota_1)^* size\ \tau_2) \textbf{ else}\\
&\textbf{if } \tau{::}[\nu{:}record{:}(I_1{:}\tau_1, \ldots I_n{:}\tau_n)] \textbf{ then}\\
&\quad ad(a[\![I_1]\!])\tau_1\ z \bigcup\\
&\quad ad(a[\![I_2]\!])\tau_2(z + size\ \tau_1) \bigcup\\
&\quad \ldots
\end{aligned}$$

5.3.3. Storage maps

We define the domain of storage maps as

$$s \in \Sigma = (L_\tau \times N) \to T.$$

A pair (a, m) is in a storage map s if the relative location a in LS corresponds to the location m in LT. For each scope a different storage map obtains.

Note, that a storage map is only defined for relative locations (L_τ), e.g. nothing is said about array objects in $I \to Lv_s$. This is justified since complex L–values $\in I \to L_v$ are not stored in the target program; only elements

2.) To be precise, relations should be defined as mappings into $\{TT, \bot\}$ with $\bot \sqsubseteq TT$. This is important to guarantee the existence of recursively defined predicates. For details see [MS76,Re74]

of arrays are stored. If LS had dynamic arrays, array descriptors would be necessary for an implementation. In this case the descriptor could be viewed as representing a complex L–value.

For given environments ς and ρ and a compile time environment y the relative storage map for the current scope is given by $\Phi_\Sigma \in U_s \to U \to Y \to \Sigma$ as

$$\begin{aligned}
\Phi_\Sigma \varsigma \rho\ y = \{(z, z) \mid \exists\, I \in Id.\, &\rho^{\#3}[\![I]\!] = level\ \rho \wedge\\
&(z, z) \in ad(\rho^{\#1}[\![I]\!])(type; \varsigma[\![I]\!])(y[\![I]\!])\}
\end{aligned}$$

Absolute storage maps are defined as $l \in \Lambda = L \times N \to T$; they relate absolute addresses of LS and LT.

For given $s \in \Sigma$ and a source and a target display the corresponding absolute storage map $l \in \Lambda$ is determined by the function $\Phi_\Lambda \in \Sigma \to X \to D \to \Lambda$ as

$$\Phi_\Lambda\ s\ \chi\delta = \{(\chi^{\#2} n\ z, \delta^{\#2} n + y) \mid n = \delta^{\#4} \wedge (z, y) \in s\}$$

Relations given by Φ_Λ only relate static locations ($\in L_s$). We assume that L_d and N are isomorphic, thus a relation $l_h \in \Lambda$ can be established once and for all.

The absolute storage map will vary during program execution. For example, exiting a procedure will change the locations that are relevant for the rest of program execution. We define a structure that is similar to displays which allows us to keep track of varying storage relations at all times.

A run time storage relation ($\in Q$) is defined as

$$\beta \in Q = (N \to Q) \times Q \times \Lambda.$$

For the dynamic and all static predecessors of the current display β defines a run time storage relation. The component Λ is the dynamic storage map that obtained at the call of the current procedure.

Entering a new procedure β has to be incremented. Such a change of β is described by $up_Q \in Q \to X \to D \to N \to \Sigma$ as

$$\begin{aligned}
up_Q\ \chi\ \delta\ \beta\ n\ s = &\ \mathbf{fix}\ \lambda\hat\beta.(f, \beta, l)\\
&\textbf{where } f = \lambda\nu.\ \textbf{if } \nu < n \textbf{ then } \beta^{\#1}\nu\\
&\qquad\qquad \textbf{else if } \nu = n \textbf{ then } \hat\beta \textbf{ else } \bot\\
&\textbf{where } l = \Phi_\Lambda \chi\delta\ s \bigcup \beta^{\#3}
\end{aligned}$$

Here, $s \in \Sigma$ is the storage map that obtains at the point of the procedure call; χ and δ are coresponding source and target displays and n is the lexical level of the procedure.

5.3.4. Relations between domains

5.3.4.1. Values

We use ϑ_V to relate values of LS and LT. Values in LT are in N; values of LS are either locations or index values ($I \subseteq N$). Since locations are related by absolute storage maps, $l \in \Lambda$ has to be argument to ϑ_V; we define

$$\vartheta_V(\epsilon, l, n) \equiv \text{ if } \epsilon \in I \text{ then } \epsilon = n \text{ else } l(\epsilon, n)$$

5.3.4.2. Stores

Depending on a particular l we can define when a store in LS is equivalent to a memory state in LT. A store σ is equivalent with a memory μ if input and output files are equal, i.e. $\sigma^{\#2} = \mu^{\#2} \wedge \sigma^{\#3} = \mu^{\#3}$, and if corresponding locations are mapped into corresponding values, i.e. $\forall a, n.l(a, n) \Rightarrow \vartheta_V(\sigma a, l, \mu\, n)$. This is not quite correct though, since we are only interested in used locations. The last component of μ specifies how much memory is used in the heap, therefore we can write $\forall a, n.l(a, n) \wedge n > \mu^{\#4} \Rightarrow \vartheta_V(\sigma a, l, \mu\, n)$. In addition all $a \in L_d$ corresponding to $n \leq \mu^{\#4}$ have to be unused. We require: $\forall a, n.l(a, n) \wedge n \leq \mu^{\#4} \Rightarrow \sigma a = unused$.

Alltogether we define

$$\begin{aligned}\vartheta_S(\sigma, l, \mu) \equiv & \sigma^{\#2} = \mu^{\#2} \wedge \\ & \sigma^{\#3} = \mu^{\#3} \wedge \\ & \forall a, n.l(a, n) \wedge n > \mu^{\#4} \Rightarrow \vartheta_V(\sigma a, l, \mu\, n) \wedge \\ & \forall a, n.l(a, n) \wedge n \leq \mu^{\#4} \Rightarrow \sigma a = unused\end{aligned}$$

5.3.4.3. Displays

Two displays χ and δ and a run time storage relation β are related if the dynamic and all static predecessors are related, if the current lexical levels are equal, and if the base address in δ is greater than all memory used in the absolute storage map in β. Displays of LT have two additional components which have no counterparts in displays of LS. The relation of these components to the source language will become apparent if we consider procedure values.

$$\begin{aligned}\vartheta_X(\chi, \beta, \delta) \equiv & \vartheta_X(\chi^{\#3}, \beta^{\#2}, \delta^{\#3}) \wedge \chi^{\#4} = \delta^{\#4} \wedge \\ & \forall n < \delta^{\#4}.\vartheta_X(\chi^{\#1} n, \delta^{\#1} n) \wedge \\ & used(\beta^{\#3}) < \delta^{\#2}\delta^{\#4}\end{aligned}$$

Here *used* determines the greatest memory location in LT that is in use according to $l \in \Lambda$. We define

$$used(l) = max\{n \mid (a, n) \in l\}$$

If a procedure is entered and χ, β and δ are incremented accordingly, the relation ϑ_X is preserved. We have the lemma

$$\vartheta_X(\chi, \beta, \delta) \Rightarrow \vartheta_X(up_S\chi\, n, up_Q\chi\delta\beta\, n\, s, up_T\delta\gamma\, n\, m\, k)$$

The proof is immeadiate by the definition of up_S, up_Q, and up_T.

5.3.4.4. Continuations

The domain in LT corresponding to generalised dynamic continuations G_d is $S \to M \to A$. We define

$$\begin{aligned}\vartheta_{G_d}(\gamma, l, \theta) \equiv & \text{ if } \gamma \in C \text{ then } \forall \sigma, \mu.\vartheta_S(\sigma, l, \mu) \Rightarrow \vartheta_A(\gamma\sigma, \theta()\mu) \\ & \text{ else } \forall \epsilon, n.\vartheta_V(\epsilon, l, n) \Rightarrow \vartheta_{G_d}(\gamma\epsilon, l, \lambda\sigma.\theta(n \cdot \sigma))\end{aligned}$$

Given a compile time environment $y \in Y$ we can define when two continuations correspond:

$$\vartheta_G(\gamma_S, s, \gamma_T) \equiv \forall \chi, \beta, \delta.\vartheta_X(\chi, \beta, \delta) \Rightarrow \vartheta_{G_d}(\gamma_S\chi, l, \gamma_T\delta)$$

where $l = \Phi_{\Lambda s}\, \chi\delta \bigcup \beta^{\#3} \bigcup l_h$. That is, l is the relation of all dynamic locations (l_h) and all used static locations.

5.3.4.5. Procedures and functions

There is no analog to procedure or function values in LT. Procedure and function values correspond to continuations in LT in the following way.

If a label j in a LT program marks the beginning of the code for a procedure, then this procedure can be called by the instruction

$$CALL\ j\ n\ m\ k$$

where n is the lexical level of the procedure, m is the number of parameters, and k is the number of memory cells allocated in the current environment.

Correspondence between a procedure identifier p and the label $j = y[\![p]\!]$ is given if a procedure call to j has the same effect as executing the procedure p. Formally we define

$$\begin{aligned}\vartheta_P(\pi, n, m, \gamma) \equiv & \forall s, \gamma_S, \gamma_T.\vartheta_G(\gamma_S, s, \gamma_T) \Rightarrow \vartheta_G(\pi\gamma_S, s, \hat{\gamma}) \\ & \text{where } \hat{\gamma} = \lambda\delta\sigma.\gamma(up\ \delta\gamma_T(n+1)(length\ \sigma - m)k)\sigma \\ & \text{where } k = used(s)\end{aligned}$$

It is important, that the value k is determined from the storage map at call time since it is used to allocate (yet unused) memory for the activation record of the current call.

5.3.4.6. Environments

The only objects bound in environments of LT are labels; there are no identifiers in LT. For ρ_S in LS to be equivalent to ρ_T in LT the following conditions must hold.

- Labels of LS are bound to continuations in ρ_s. For each label the compile time environment y gives a corresponding label of LT. Suppose, we have label N in LS bound to $\gamma = \rho_S[\![N]\!]$. Assume that according to y label N corresponds to $j = y[\![N]\!]$ in LT. Then we require that γ and $\rho_T[\![j]\!]$ are equivalent with respect to ϑ_G.
- Procedure identifiers in LS are bound to procedure values in ρ_S. In the compile time environment a label in LT is assigned to each procedure identifier. The continuation bound to this label in ρ_T has to correspond to the procedure value according to ϑ_P.

$$\begin{aligned}\vartheta_U(\rho_S,\varsigma,y,\rho_T) \equiv & \forall N \in Num.\vartheta_G(\rho_s^{\#4}[\![N]\!],s,\rho_T[\![y[\![N]\!]]\!]) \wedge \\ & \textbf{where } n = \rho_S^{\#5}[\![I]\!] \\ & \forall I \in Id.\vartheta_P(\rho_S^{\#2}[\![I]\!],n,m,\rho_T[\![y[\![I]\!]]\!]) \\ & \textbf{where } n = \rho_S^{\#3}[\![I]\!] \\ & \textbf{where } s = \Phi_\Lambda\varsigma\rho\, y \\ & \textbf{where } \varsigma[\![I]\!]::[proc{:}r_1,\ldots,r_m]\end{aligned}$$

5.3.5. Existence of recursive predicates

The definition of the predicates is cyclic or recursive. Therefore we have to investigate whether these relations are at all well defined. We did not have similar problems with recursive functions since our formal language syntactically guarantees that all functions are continuous and for continuous functions least fixed points always exist.

The language used to define predicates ϑ_i is a significant extension of Scott's logic of computable functions. In particular set constructors are used freely. Therefore there is no syntactic guarantee that the so defined relations (functions into T) are continuous. In fact, the predicates defined above are not

even monotonic. For example, we have

$$\begin{aligned}\vartheta_S(\sigma,l,\mu) \equiv & \sigma^{\#2} = \mu^{\#2} \wedge \\ & \sigma^{\#3} = \mu^{\#3} \wedge \\ & \forall a,n.l(a,n) \wedge n > \mu^{\#4} => \vartheta_V(\sigma a,l,\mu\, n) \wedge \\ & \forall a,n.l(a,n) \wedge n \leq \mu^{\#4} => \sigma a = unused\end{aligned}$$

If l increases ϑ_S becomes smaller.

The above situation has been discussed extensively in the literature [MS76, Re74, St77]. It has been shown that under certain conditions recursively defined predicates have a least fixed point even if they are not monotonic. To repeat the theory of "inclusive" predicate is beyond the scope of this thesis. However, Reynolds [Re74] presents a set of easy to check criteria that ensure existence of fixed points. It can be seen that our definitions satisfy his criteria; thus we subsequently assume that the above predicates are well defined.

5.4. Implementation of the code generation

5.4.1. Specifying code generating procedures

The relations introduced above can be used to specify the correctness of our recursive code generating procedures. For example, *Ecode* generating code for expressions, is specified as

```
procedure Ecode(E: asyn; ς: U_s; ρ: U_S; y: U_C; ρ_T:U_T; var z:code);
initial z = z0;
exit ∀γ, γ_T.(ϑ_U(ρ, ς, y, ρ_T) ∧ ϑ_G(γ, Φ_Σςρ y, M[[z0]]ρ_T γ_T)) =>
    ϑ_G(ℰ[[E]]ςργ, Φ_Σςρ y, M[[z]]ρ_T γ_T);
```

Note, that this specification assumes that there is some initial sequence of code $z0$ to which the new code is appended, yielding z.

One problem with this specification is that it envolves quantification. Our solution is to introduce auxiliary parameters to allow instantiation of universally quantified variables as outlined in chapter II. For each expression we know the source continuation which applies. If we also knew the target continuation we could provide the correct instances of these continuations with each call to *Ecode* and thus avoid quantification. The specification would then be

```
procedure Ecode(E: asyn; ς: U_s; ρ: U_S; γ: G_S;
                y: U_C; ρ_T:U_T; γ_T:G_T; var z:code);
entry ϑ_U(ρ, ς, y, ρ_T) ∧ ϑ_G(γ, Φ_Σςρ y, M[[z]]ρ_T γ_T);
exit ϑ_G(ℰ[[E]]ςργ, Φ_Σςρ y, M[[z]]ρ_T γ_T);
```

...currence of z in the entry conditions refers to the initial ... z in the exit condition refers to the final code.

... the instantiation technique we have to know the correct in... target continuation. But this in turn requires that we know the ... follows the expression we are translating. For this reason we decide ... rate code backwards! That is, the code generation will produce code in ...erse order, last instruction first.

Let use now look at the definition of *AEcode* (*AEcode* corresponds to $\hat{\mathcal{E}}$ ...t has specifications identical to *Ecode*). Some typical cases are

```
procedure AEcode(E: asyn;
                 zeta: static_environment;
                 rho: Tenvironment;
                 gamma: Scontinuation;
                 y: compile_environment;
                 rhoT: Tenvironment;
                 gammaT: Tcontinuation;
                  var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiL(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(AEscr(E, zeta, rho, gamma), PhiL(zeta, rho, y),
        Mscr(z, rhoT, gammaT));
var E1, E2, Elist, I, N, O: asyn;
    alpha: location;
    epsilon: value;
    mu, mulist: mode;
    nu, m: integer;
    tau, tau0, tau1: ttype;
begin
  if isnumber(E) then
  begin
    matchnumber(E, N);
    cmkelit(Nscr(N), rhoT, gammaT, z);
  end else
  if isexnid(E) then
```



```
    mu ← ce(E1, zeta);
    tau ← ctype(mu);
    if isarraytype(tau) then
    begin
      matcharraytype(tau, nu, tau0, tau1);
      indexcode(zeta, rho, gamma, y, rhoT, gammaT, z);
      Acode(E2, zeta, mkevalmode(tau0), rho, indez(gamma), y, rhoT, gammaT, z);
      Ecode(E1, zeta, rho, Ascr(E2, zeta, mkevalmode(tau0), rho,
          indez(gamma)), y, rhoT, gammaT, z);
    end else error
  end else
  if isselect(e) then
  begin
    matchselect(E, E1, I);
    selectcode(I, zeta, rho, gamma, y, rhoT, gammaT, z);
    Ecode(E1, zeta, rho, select(I, gamma), y, rhoT, gammaT, z);
  end else error;
end ;
```

Most of the details of generating code are delegated to auxiliary functions such as *selectcode*, *indexcode*, *cmkelit* and so on. Typical specifications of these functions are

```
procedure cmkelit(epsilon:value; rhoT:Tenvironment;
                  gammaT:Tcontinuation; var z:code);
initial z = z0;
entry true;
exit  Mscr(z, rhoT, gammaT) = Tapply(Mscr(z0, rhoT, gammaT), epsilon);
external ;

procedure cmkeunop(O:asyn; rhoT:Tenvironment;
                   gammaT:Tcontinuation; var z:code);
initial z = z0;
entry true;
exit  Mscr(z, rhoT, gammaT) = Tunop(Mscr(z0, rhoT, gammaT), O);
external ;
```

Let us explain *cmkelit*, which generates an instruction $LIT\ \epsilon$ and appends

The proof of the verification conditions for *AEcode* is immediate from the definition of $\hat{\mathcal{E}}$ and some to be defined properties of the predicates ϑ_i. Consider the following example for the case where E is a numeral.

$$
\begin{array}{l}
(varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT)) \wedge \\
\quad varthetaU(rho, zeta, y, rhoT) \wedge \\
\quad isnumber(e) \wedge \\
\quad e = mkenumber(n\_0) \wedge \\
\quad Mscr(z\_0, rhoT, gammaT) = Tapply(Mscr(z, rhoT, gammaT), Nscr(n\_0)) \\
\quad => \\
\quad\quad varthetaG(AEscr(e, zeta, rho, gamma), PhiS(zeta, rho, y), \\
\quad\quad\quad Mscr(z\_0, rhoT, gammaT)))
\end{array}
$$

For the system's prover this verification condition poses no problem, given a suitable axiomatization of the concepts involved (see appendix 6.). Let us demonstrate the validity of the above condition manually to illustrate the basic schema of proof used to verify the code generation. It is advantageous to rewrite the verification condition in a shorter mathematical notation.

$$
\begin{array}{l}
\vartheta_G(\gamma, \Phi_\Sigma \varsigma \rho\ y, \mathcal{M}[\![z]\!]\rho_T\gamma_T) \wedge \\
\vartheta_U(\rho, \varsigma, y, \rho_T) \wedge \\
\mathcal{M}[\![z\_0]\!]\rho_T\gamma_T = \lambda\delta\sigma.\mathcal{M}[\![z]\!]\rho_T\gamma_T\delta(n{\cdot}\sigma) \\
=> \\
\quad \vartheta_G(\hat{\mathcal{E}}[\![N]\!]\varsigma\rho\gamma, \Phi_\Sigma\varsigma\rho\ y, \mathcal{M}[\![z\_0]\!]\rho_T\gamma_T)
\end{array}
$$

Substituting $\hat{\gamma}_T$ for $\mathcal{M}[\![z]\!]\rho_T\gamma_T$ and expanding the definition of $\hat{\mathcal{E}}$ we have to prove (omitting some redundant assumptions)

$$
\begin{array}{l}
\vartheta_G(\gamma, s, \hat{\gamma}_T) \wedge \\
=> \\
\quad \vartheta_G(apply\ \gamma(\mathcal{N}[\![N]\!]), s, \lambda\delta\sigma.\hat{\gamma}_T\delta(\mathcal{N}[\![N]\!]{\cdot}\sigma))
\end{array}
$$

This formula is a theorem and can be added to the logical basis of the verification. It can be proven as follows. By definition of $\vartheta_G(\gamma, s, \hat{\gamma}_T)$ we get

$$\forall\chi\beta\delta.\vartheta_X(\chi, \beta, \delta) => \vartheta_{G_d}(\gamma\chi, l, \hat{\gamma}_T\delta)$$

for $l = \Phi_\Lambda s\ \chi\delta \bigcup \beta^{\#3} \bigcup l_h$.

Since we have $\vartheta_V(\mathcal{N}[\![N]\!], l, \mathcal{N}[\![N]\!])$ the definition of ϑ_{G_d} yields

$$\vartheta_{G_d}(\gamma\chi, l, \hat{\gamma}_T\delta) => \vartheta_{G_d}(\gamma\chi\mathcal{N}[\![N]\!], l, \lambda\sigma.\hat{\gamma}_T\delta(\mathcal{N}[\![N]\!]{\cdot}\sigma))$$

for any χ, β, and δ satisfying $\vartheta_X(\chi, \beta, \delta)$. By definition this means that

$$\vartheta_G(apply\ \gamma(\mathcal{N}[\![N]\!]), s, \lambda\delta\sigma.\hat{\gamma}_T\delta(\mathcal{N}[\![N]\!]{\cdot}\sigma))$$

holds.

Similar theorems about properties of ϑ_i are required for other cases of *Ecode*. Prove of these theorems is not possible within the verifier as it involves quantification.

Other valuations follow the schema outlined for $\mathcal{E}$. However, there are some points that require some further attention.

- What happens to labels? How does the fixed point describing the meaning of labels relate to the produced code?
- How are declarations treated?
- How are procedures translated?

5.4.2. Treatment of labels

The meaning of a sequence of statements is defined by $C\ell$ as a least fixed point binding labels to continuations. For the code generating procedure corresponding to $C\ell$ we have to prove that the code produced for a list of statements has the correct meaning. To do this we produce a block. The meaning of this block is given as a least fixed point binding labels to target continuations. Thus we have to prove the equivalence of two fixed points, $fix\ f_S$ and $fix\ f_T$. More precisely, corresponding to $C\ell$ we use a procedure *Clcode* to produce code for statement sequences. For given y and ς we want to prove that *Clcode* satisfied the entry-exit specification:

$$
\begin{array}{l}
\textbf{entry}\ \vartheta_U(\rho, \varsigma, y, \rho_T) \wedge \vartheta_G(\gamma, \Phi_\Sigma\varsigma\rho\ y, \gamma_T); \\
\textbf{exit}\ \vartheta_G(C\ell[\![\Gamma]\!]\varsigma\rho\gamma, \Phi_\Sigma\varsigma\rho\ y, \mathcal{M}[\![z]\!]\rho_T\gamma_T);
\end{array}
$$

where γ and γ_T are the correct instances of the source and target continuations.

According to the semantics of LS and LT we have

$$
\begin{array}{l}
C\ell[\![\Gamma]\!]\varsigma\rho\gamma = C[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma \\
\qquad\qquad \text{where } \hat{\varsigma} = \varsigma\big[(\ldots, true, \ldots)/j[\![\Gamma]\!]\big], \\
\qquad\qquad \hat{\rho} = fix\Big(\lambda\hat{\rho}.\rho\big[J[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma/j[\![\Gamma]\!]\big]\Big)
\end{array}
$$

and for $z = (I_1 \ldots I_n)$

$$\mathcal{M}[\![z]\!]\rho_T\gamma_T = \mathcal{M}^*[\![I_1 \ldots I_n]\!]\rho_1\gamma$$

where $\rho_1 = fix(\lambda\rho.\rho_T[\mathcal{L}[\![I_1 \ldots I_n]\!]\rho\gamma_T/l[\![I_1 \ldots I_n]\!]])$

Assume y and ς are fixed for now and suppose that we can prove the

following property for $I_1, \ldots, I_n$.

$$\forall\, \bar{\rho}\ \bar{\rho}_T\ \bar{\gamma}\ \bar{\gamma}_T.(\vartheta_U(\bar{\rho}, \varsigma, y, \bar{\rho}_T) \wedge \vartheta_G(\bar{\gamma}, \Phi_\Sigma \varsigma\ \bar{\rho}\ y, \bar{\gamma}_T) \Rightarrow$$
$$\vartheta_G(C[\![\Gamma]\!] \varsigma\ \bar{\rho}\ \bar{\gamma}, \Phi_\Sigma \varsigma\ \bar{\rho}\ y, M[\![I_1, \ldots, I_n]\!]\ \bar{\rho}_T\ \bar{\gamma}_T) \wedge$$
$$\vartheta_G^*(J[\![\Gamma]\!] \varsigma\ \bar{\rho}\ \bar{\gamma}, \Phi_\Sigma \varsigma\ \bar{\rho}\ y, \mathcal{L}[\![I_1, \ldots, I_n]\!]\ \bar{\rho}_T\ \bar{\gamma}_T))$$

where ϑ_G^* is true for two lists of continuations if corresponding elements are related by ϑ_G. Assuming that the initial environments are related (entry condition), i.e. $\vartheta_U(\rho, \varsigma, y, \rho_T)$, then by fixed point induction it is immediate that $\vartheta_U(\hat{\rho}, \varsigma, y, \rho_1)$. Assuming further, that the initial continuations are related (entry condition), i.e. $\vartheta_G(\gamma, \Phi_\Sigma \varsigma \rho\ y, \gamma_T)$, the exit condition follows.

In the above arguments we assumed that the predicate ϑ_U is admissible. In fact, Reynold's theorem [Re74] mentioned earlier not only guarantees the existence of the recursive predicates ϑ_i but also proves that these predicates are admissible[3].

5.4.2.1. Implementation of *Clcode*

We assume a procedure *Ccode*, corresponding to C, with the specification

procedure *Ccode*(Γ:*asyn*; ς:U_s; *y*:*Y*; **var** *z*:*code*);
exit $\forall \rho \rho_T \gamma \gamma_T.\vartheta_U(\rho, \varsigma, y, \rho_T) \wedge \vartheta_G(\gamma, \Phi_\Sigma \varsigma \rho\ y, \gamma_T) \Rightarrow$
$\vartheta_G(C[\![\Gamma]\!] \varsigma \rho \gamma, \Phi_\Sigma \varsigma \rho\ y, M[\![z]\!] \rho_T \gamma_T) \wedge \vartheta_G^*(J[\![\Gamma]\!] \varsigma \rho \gamma, \Phi_\Sigma \varsigma \rho\ y, \mathcal{L}[\![z]\!] \rho_T \gamma_T) \wedge$
$\ell[\![z]\!] = y\ j[\![\Gamma]\!]$;

Procedure *Clcode* can then be written as

procedure *Clcode*(Γ:*asyn*; ς:U_s; ρ:*U*; γ:*Gs*;
y:*Y*; ρ_T:U_T; γ_T:G_T; **var** *z*:*code*);
entry $\vartheta_U(\rho, \varsigma, y, \rho_T) \wedge$
$\vartheta_G(\gamma, \Phi_\Sigma \varsigma \rho\ y, M[\![z]\!] \rho_T \gamma_T)$;
exit $\vartheta_G(C\ell[\![\Gamma]\!] \varsigma \rho \gamma, \Phi_\Sigma \varsigma \rho\ y, M[\![z]\!] \rho_T \gamma_T)$;

var *znew*:*code*;
ς_p:U_s;
ynew:*Y*;

begin
$\varsigma_p \leftarrow \varsigma\big[(\ldots, true, \ldots)/j[\![\Gamma]\!]\big]$
mkeTlabels(*y*, *j*[⟦Γ⟧], *ynew*);
Ccode(Γ, ς_p, *ynew*, *znew*);
cmkeblockcode(*znew*, *z*, ρ_T, γ_T);
end ;

3.) Inclusive predicates are admissible

mkeTlabels generates a new set of labels for *LT* corresponding to *j*⟦*G*⟧ and enters this correspondence in *y*, resulting in *ynew*.

The procedure *cmkeblockcode* converts the list of instructions *znew* into a block (*znew*) and appends this block to the front of *z*. The parameters ρ_T and γ_T are virtual and only required in the entry–exit specifications.

procedure *cmkeblockcode*(*znew*:*code*; **var** *z*:*code*; ρ_T:U_T; γ_T:G_T);
initial $z = z0$;
exit $M[\![z]\!]\rho_T\gamma_T = M[\![Tmkeblock(znew)]\!]\rho_T; M[\![z0]\!]\rho_T\gamma_T$;

For the above implementation of *Clcode* the following verification condition has to be proven.

$$\vartheta_U(\rho, \varsigma, y, \rho_T) \wedge \vartheta_G(\gamma, \Phi_\Sigma \varsigma \rho\ y, M[\![z]\!] \rho_T \gamma_T) \wedge$$
$$distinct(\dot{y}\ j[\![G]\!]) \wedge$$
$$\forall \dot{\rho} \dot{\rho}_T \dot{\gamma} \dot{\gamma}_T.(\vartheta_U(\dot{\rho}, \dot{\varsigma}, \dot{y}, \dot{\rho}_T) \wedge \vartheta_G(\dot{\gamma}, \Phi_\Sigma \dot{\varsigma} \dot{\rho}\ \dot{y}, \dot{\gamma}_T) \Rightarrow$$
$$\vartheta_G(C[\![G]\!] \dot{\varsigma} \dot{\rho} \dot{\gamma}, \Phi_\Sigma \dot{\varsigma} \dot{\rho}\ \dot{y}, M[\![\dot{z}]\!] \dot{\rho}_T \dot{\gamma}_T) \wedge$$
$$\vartheta_G^*(J[\![G]\!] \dot{\varsigma} \dot{\rho} \dot{\gamma}, \Phi_\Sigma \dot{\varsigma} \dot{\rho}\ \dot{y}, \mathcal{L}[\![\dot{z}]\!] \dot{\rho}_T \dot{\gamma}_T)) \wedge$$
$$\ell[\![\dot{z}]\!] = \dot{y}\ j[\![G]\!]$$
$$\Rightarrow$$
$$\vartheta_G(C\ell[\![G]\!] \varsigma \rho \gamma, \Phi_\Sigma \varsigma \rho\ y, M[\![(\dot{z}); z]\!] \rho_T \gamma_T);$$
$$\text{where } \dot{\varsigma} = \varsigma\big[(\ldots, true, \ldots)/j[\![G]\!]\big]$$
$$\text{where } \dot{y} = y\big[labs/j[\![G]\!]\big]$$
$$\text{for new unique labels } labs$$

We let $\dot{\gamma}_T = M[\![z]\!]\rho_T\gamma_T$. If we apply the definitions of $C\ell$ and M to the conclusion of the VC we get

$$\vartheta_G(C[\![G]\!] \dot{\varsigma} \dot{\rho} \gamma, \Phi_\Sigma \varsigma \rho\ y, M[\![\dot{z}]\!] \rho_T \dot{\gamma}_T)$$
$$\text{where } \dot{\rho} = \mathbf{fix}\ (\bar{\rho}.\rho\big[J[\![G]\!] \dot{\varsigma} \bar{\rho} \gamma / j[\![G]\!]\big])$$
$$\text{where } \dot{\rho}_T = \mathbf{fix}\ (\bar{\rho}_T.\rho_T\big[\mathcal{L}[\![\dot{z}]\!] \bar{\rho}_T \dot{\gamma}_T / \dot{y}\ j[\![G]\!]\big])$$

Since locations bound in ς, ρ, and *y* are identical to those in $\dot{\varsigma}$, $\dot{\rho}$, and $\dot{y}$ we have

$$\Phi_\Sigma \varsigma \rho\ y = \Phi_\Sigma \dot{\varsigma} \dot{\rho}\ \dot{y}$$

Now, the proof of this verification condition follows by fixed point induction as outlined above.

5.4.2.2. Weak ∀-introduction

The remaining problem is to prove the universally quantified exit condition of *Ccode*.

$$\forall \rho\rho_T\gamma\gamma_T.\vartheta_U(\rho,\varsigma,y,\rho_T)\wedge \vartheta_G(\gamma,\Phi_\Sigma\varsigma\rho\ y,\gamma_T) => \vartheta_G(C[\![G]\!]\varsigma\rho\gamma,\Phi_\Sigma\varsigma\rho\ y,\mathcal{M}[\![z]\!]\rho_T\gamma_T)\wedge \vartheta_G^{\cdot}(J[\![G]\!]\varsigma\rho\gamma,\Phi_\Sigma\varsigma\rho\ y,\mathcal{L}[\![z]\!]\rho_T\gamma_T)$$

The solution is weak ∀-introduction defined in chapter II. We prove that the body of *Ccode* satifies the following quantifier free specifications.

```
procedure Ccode(G:asyn; ς:U_s; ρ:U; γ:G;
                y:Y; ρ_T:U_T; γ_T:U_T; var z:code);
entry ϑ_U(ρ, ς, y, ρ_T) ∧ ϑ_G(γ, Φ_Σ ς ρ y, γ_T);
exit  ϑ_G(C⟦G⟧ς ρ γ, Φ_Σ ς ρ y, M⟦z⟧ρ_T γ_T) ∧ ϑ_G^·(J⟦G⟧ς ρ γ, Φ_Σ ς ρ y, L_T⟦z⟧ρ_T γ_T);
```

The rule of weak ∀-introduction allow to use the specifications

```
entry true;
exit  ∀ρρ_T γγ_T.ϑ_U(ρ, ς, y, ρ_T) ∧ ϑ_G(γ, Φ_Σ ς ρ y, γ_T) =>
        ϑ_G(C⟦G⟧ς ρ γ, Φ_Σ ς ρ y, M⟦z⟧ρ_T γ_T) ∧ ϑ_G^·(J⟦G⟧ς ρ γ, Φ_Σ ς ρ y, L⟦z⟧ρ_T γ_T);
```

for calls to *Ccode*. The universally quantified exit condition can be abbreviated as

$$forallUGG(G,\varsigma,y,z) \equiv \forall \rho\rho_T\gamma\gamma_T.\vartheta_U(\rho,\varsigma,y,\rho_T)\wedge \vartheta_G(\gamma,\Phi_\Sigma\varsigma\rho\ y,\gamma_T) => \vartheta_G(C[\![G]\!]\varsigma\rho\gamma,\Phi_\Sigma\varsigma\rho\ y,\mathcal{M}[\![z]\!]\rho_T\gamma_T)\wedge \vartheta_G^{\cdot}(J[\![G]\!]\varsigma\rho\gamma,\Phi_\Sigma\varsigma\rho\ y,\mathcal{L}[\![z]\!]\rho_T\gamma_T)$$

in our assertion language.

We have shown above that by fixed point induction it is possible to prove

$$\begin{array}{l}\vartheta_U(\rho,\varsigma,y,\rho_T)\wedge\vartheta_G(\gamma,\Phi_\Sigma\varsigma\rho\ y,\mathcal{M}[\![z]\!]\rho_T\gamma_T)\wedge\\ distinct(\dot{y}\ j[\![G]\!])\wedge\\ forallUGG(G,\dot{\varsigma},\dot{y},\dot{z})\wedge\\ \ell[\![\dot{z}]\!]=\dot{y}\ j[\![G]\!]\\ =>\\ \quad \vartheta_G(C\ell[\![G]\!]\varsigma\rho\gamma,\Phi_\Sigma\varsigma\rho\ y,\mathcal{M}[\![(\dot{z});z]\!]\rho_T\gamma_T);\\ \quad \text{where } \dot{\varsigma}=\varsigma[(\ldots,true,\ldots)/j[\![G]\!]]\\ \quad \text{where } \dot{y}=y[labs/j[\![G]\!]]\\ \quad\quad \text{for new unique labels } labs\end{array}$$

This theorem can now be added to our logical basis as a rule

```
infer varthetaG(Cscrl(G, zeta, rho, gamma), s, Mscr(Tmkeblock(z), rhoT, gammaT))
  from ForallUGG(G, redeftrue(zeta, j(G)), yy, z) ∧
    varthetaU(rho, zeta, y, rhoT) ∧
    s = PhiS(zeta, rho, y) ∧
    disjointlabels(yy, j(G), y) ∧
    VarthetaG(gamma, s, gammaT);
```


5.4.3. Declarations

There are code generating procedures corresponding to valuations that deal with declarations (e.g. $\mathcal{V}$, $\mathcal{Q}$). But these procedures will not generate code, rather they will change the compile time environment by allocating new storage.

For example we have

$$\mathcal{V}_v[\![I{:}T]\!]\varsigma\rho = \text{if } \lfloor var{:}\tau\rfloor{::}\varsigma[\![I]\!] \text{ then } \dot{\rho}[a/I] \text{ where } (a,\dot{\rho}) = newl\ \tau\rho.$$

The corresponding procedure *Allocate* will allocate memory for I in LT.

```
procedure Allocate(D:asyn; ς:U_s; ρ:U; var y:Y; ρ_T:U_T);
entry ϑ_U(ρ, ς, y, ρ_T);
exit ϑ_U(V_v⟦I:T⟧ς ρ, ς, y, ρ_T);

begin
  if ⌊var:τ⌋::ς⟦I⟧ then
  begin
    n ← size(τ);
    y ← ⟨y^#1 | y^#4/I⟩, y^#2, y^#3, y^#4 + n);
  end
end ;
```

A proof of this procedure is immediate, considering that no continuation or procedure value in ρ or ρ_T have been changed.

Meaning functions that describe initialization of variables are similar to other executable statements; their implementation follows analogously.

5.4.4. Procedures and functions

We mentioned above that there are no procedure and function values in LT. A procedure in LS corresponds to a label in LT, the start address of the code for this procedure. The relation ϑ_U guarantees that for any procedure value π and the associated continuation γ of the start address of the procedure the following holds:

$$\forall\gamma_S,\gamma_T.\vartheta_G(\gamma_S,s,\gamma_T) => \vartheta_G(\pi\gamma_S,s,\hat{\gamma}) \quad \text{where } \hat{\gamma}=\lambda\delta\sigma.\gamma(up\ \delta\gamma_T n(length\ \sigma - m)k)\sigma$$

Here, n is the lexical level of the definition of the procedure, m is the number of parameters, and k is the number of cells allocated in the calling environment.

Since the semantics of the *CALL* instruction is given by

$$M[\![CALL\ l\ n\ m\ k]\!]\rho\gamma = \lambda\delta\sigma.\rho[\![l]\!](up\ \delta\ \gamma\ n(length\ \sigma - m)k)\sigma$$

the correctness of code for a procedure call is immediate. The necessary parameters to *CALL* are readily available at call time. k is given as $y^{\#4}$.

It remains to be shown that correct code is generated for a procedure declaration. We have the definition

$$\begin{aligned}\mathcal{F}[\![&procedure\ I(\Pi_1,\ldots,\Pi_n);\Theta]\!]\varsigma\rho = \langle\pi\rangle\\ &\text{where } \pi = \lambda\gamma.enter(n+1);P^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1\rho_1;\\ &\qquad B[\![\Theta]\!]\varsigma_2\rho_1;exit\ \gamma\\ &\text{where } \varsigma_2 = \varsigma_1\big[\lfloor proc{:}\mu_1,\ldots,\mu_n\rfloor/I\big]\\ &\text{where } \langle(\mu_1,\ldots,\mu_n),\varsigma_1\rangle = p^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma\\ &\text{where } \rho_1 = \mathcal{Q}^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1(next\ \rho)\\ &\text{where } n = level\ \rho\end{aligned}$$

Code for such a declaration is generated by a procedure *Fcode* with the following specifications.

procedureFcode(*P*:*asyn*, ς:U_s, ρ:U, y:Y, γ_T:G_T, ρ_T:U_T, var *z*:*code*);
exit $\vartheta_P(\mathcal{F}[\![P]\!]\varsigma\rho, level\ \rho, parms(P,\varsigma), Mscr[\![z]\!]\rho_T\gamma_T)$;

where *parms* returns the number of parameters of P in the environment ς.

Fcode cenerates the following code.

- *Qcode* is called which changes the compile time environment to contains bindings of parameters to target locations. Let $\dot{y}$ be this new compile time environment.
- *Pcode* generates code z_P that initializes parameters with the values of actual parameters. These values are all on top of the stack (in reverse order).
- *Bcode* produces code z_B for the procedure body.
- The *EXIT* instruction causes return from the procedure.

According to the semantics of *LT* we have $M[\![EXIT]\!]\rho_T\gamma_T = \lambda\delta.\delta^{\#5}\delta^{\#3}$. Thus, for the final code z produced by *Fcode* we have

$$M[\![z]\!]\rho_T\gamma_T = M[\![z_P;z_B]\!]\rho_T;\lambda\delta.\delta^{\#5}\delta^{\#3}.$$

By the source semantics we have

$$\mathcal{F}[\![P]\!]\varsigma\rho = \lambda\gamma.enter(n+1);P^*[\![\Pi_i]\!]\varsigma_1\rho_1;B[\![\Theta]\!]\varsigma_2\rho_1;exit\ \gamma$$

where ς_1, ς_1, and ρ_1 are defined as in the definition above; $n = level\ \rho$.

Expanding the exit condition of *Fcode*, we have to prove that

$$\begin{aligned}&\forall s,\gamma_S,\dot{\gamma}_T.\vartheta_G(\gamma_S,s,\dot{\gamma}_T) => \vartheta_G(\overline{\gamma}_S,s,\overline{\gamma}_T)\\ &\text{where } \overline{\gamma}_S = enter(n+1);P^*[\![\Pi_i]\!]\varsigma_1\rho_1;B[\![\Theta]\!]\varsigma_2\rho_1;exit\ \gamma_S\\ &\text{where } \overline{\gamma}_T = \lambda\delta\sigma.M[\![z_P;z_B]\!]\rho_T(\lambda\delta.\delta^{\#5}\delta^{\#3})(up\ \delta\dot{\gamma}_T\ n(length\ \sigma - m)k)\sigma\end{aligned}$$

where $k = used(s)$.

To prove this verification condition we use the fact that

$$M[\![z]\!]\rho_T\gamma_T\delta = g[\![z]\!]\rho_T(\gamma_T\delta)\delta$$

for some function g. This formula is not valid for arbitrary pieces of code. However, it holds for all code sequences z produced by our code generating functions.

Now, consider $M[\![z]\!]\rho_T(\lambda\delta.\delta^{\#5}\delta^{\#3})\dot{\delta}$. By the above lemma we have

$$M[\![z]\!]\rho_T(\lambda\delta.\delta^{\#5}\delta^{\#3})\dot{\delta} = g[\![z]\!]\rho_T((\lambda\delta.\delta^{\#5}\delta^{\#3})\dot{\delta})\dot{\delta} =$$

$$g[\![z]\!]\rho_T((\lambda\delta.\dot{\delta}^{\#5}\delta^{\#3})\dot{\delta})\dot{\delta} = M[\![z]\!]\rho_T(\lambda\delta.\dot{\delta}^{\#5}\delta^{\#3})\dot{\delta}$$

Applied to the verification condition for *Fcode* we can rewrite $\overline{\gamma}_T$ as

$$\overline{\gamma}_T = \lambda\delta\sigma.M[\![z_P;z_B]\!]\rho_T;\lambda\delta.\dot{\gamma}_T\delta^{\#3}(up\ \delta\gamma_T\ n(length\ \sigma - m)k)\sigma$$

Thus, we have shown, that the procedure returns to the right point.

The proof of the verification condition now follows from the correctness of the components z_P and z_B. With $\vartheta_G(\gamma_S,s,\dot{\gamma}_T)$ we get $\vartheta_G(exit\ \gamma_S,\dot{s},\lambda\delta.\dot{\gamma}_T\delta^{\#3})$. From this and the correctness of z_B we get

$$\vartheta_G(B[\![\Theta]\!]\varsigma\rho;exit\ \gamma_S,\dot{s},M[\![z_B]\!]\rho_T(\lambda\delta.\dot{\gamma}_T\delta^{\#3}))$$

and so on.

5.4.5. Blocks

The meaning of a block in *LS* is defined by a least fixed point similar to $C\ell$. The main difference is that not only labels are bound in the environment by also procedures and functions are bound to their procedure and function values. The implementation strategy for blocks is the same as that for command lists. Weak $\forall$-introduction is used in the very same way as for *Clcode*.

5.4.6. Refinement

We have described the overall strategy as well as all of the non obvious parts of the code generation. The remaining refinement steps follow along the

same line as for static semantics. We use the same representations for the abstract syntax, type, modes, and environments.

We do not refine the representation of code sequences any further here. Using the techniques outlined earlier this step should be simple given a concrete representation for the machine code.

Some typical code generation procedures are given in appendix 6.

Chapter V. Conclusions

1. Summary

We have presented the theory, specification, implementation, and correctness proof of a compiler. The source and target languages LS and LT are realistic and useful languages rather than toys.

We have shown that a correct compiler can be systematically developed from a formal definition of source and target languages. The techniques employed in this thesis are very general and will also be applicable to other large software systems.

We have given a formal denotational definition of the source language LS on a low level of abstraction. This definition captures and formalizes standard compiling techniques. Future compilers will profit from these results whether or not they are formally verified.

In the course of the proofs we had to tackle various technical problems, such as the treatment of pointer operations, quantification in a quantifier free assertion language and the treatment of fixed points. Though apparently minor, these problems are of general importance and will most certainly arise in totally different applications.

We hope that this work is a convincing argument, that today's verification techniques are sufficiently powerful to be used in real life software development. At the same time this work reveals some of the weak points of the technology and points to future areas of research; we will discuss some possible improvements in the following sections.

Certainly there is no guarantee that our compiler will never fail. We have discussed some of the possible sources of errors earlier. Still, software developed from formal specifications and formally verified increases our confidence in its correctness to a point not achievable with conventional testing techniques alone. As the development of techniques for program verification will continue future proofs will leave less and less margin for errors.

Our compiler is not yet a "software product"; several features have to be added to make it a useful compiler. A suitable error handling and recovery mechanism as well as output of the produced code in suitable format have to be added. As we pointed out earlier, these additions can be made without invalidating the program proofs given. Furthermore, it is desirable to include a minimum of code optimization in the compiler. We will discuss some of the possibilities for this as well as other useful extensions in the following section.

Numerous programs are written today for which correctness is imperative. We believe that in many cases verification is economically feasible and that these programs would greatly benefit from verification.

2. Extensions

It is desirable to be able to verify compilers that involve code optimizations. This is particularly important since a large number of the errors of present compiler are in the optimization part of the compiler.

Also, one may wish to apply our techniques to languages with features not present in *LS*. Both points are discussed in this section.

2.1. Optimization

Except for compile time evaluation of constant expressions our compiler does not perform optimizations of any kind. Let us briefly consider how possible optimizations would affect our proofs.

Let us classify optimizations in those which require data flow analysis and those which do not. In both cases we can distinguish optimizations performed on the source and the target level.

Optimizations that require data flow analysis are very hard to include in our compiler. They require extensive proofs that certain manipulations of the source or target text do not alter the meaning of a program. The correctness of these transformations may depend on an arbitrarily large context. The necessary theorems cannot be proven (or even formulated) in our assertion language. A reasonable approach would be to include an additional "optimization" step in the compiling process and develop a suitable logical basis for it. But it should be expected that a sophisticated optimization alone approaches or exceeds the complexity of the complete compiler presented here.

Optimizations that do not require flow analysis present a much brighter picture. For example, given the sequence of assignments $x \leftarrow a; x \leftarrow b$ where

a and b are free of side effects (say simple variables) then we can omit the first assignment without changing the meaning of the program. Sequences of this nature can be detected easily and the proof that certain optimizations preserve the meaning of the program is easy. Unfortunately, redundant operations are not very common in the source langauge.

By far the easiest optimization to include in our compiler is a technique called "peephole optimization" introduced by McKeeman [Mc65]. The idea is to find redundant operations in the produced target code or to find sequences of code that can be replaced by shorter code. The idea is very simple. One moves a window across the final code and matches the instructions in the window with a given set of patters for which a correctness preserving transformation is known.

The technique will be very successful since our code generation is very primitive. Code for components of the source program is produced irrespective of the context. For example, code sequences of the form $HOP\ n; LABSET\ n$ are very common. Also, our instruction set was chosen to be minimal. Most machines have instructions that are not strictly necessary but speed up certain frequent operations. *LT* could have an instruction *INCR* which increments the top of the stack. Then the optimizer could search for the pattern $LIT\ 1; ADD$ and replace it by *INCR*.

The verification of a peephole optimizer is fairly straightforward. One can prove the validity of a set of transformations of the form $pattern_1 \rightarrow pattern_2$ and systematically apply them.

2.2. Register machines

Generating code for a register machine is substantially more complicated than producing code for a stack machine. Still, a sensible approach would be to first produce intermediate code for a stack machine and in a later translation step to convert it into code for a register machine. This later translation can be isolated from the rest of the compiler in which case a correctness proof becomes manageable. Sophisticated code for a register machine will require optimizations (e.g. register allocation) and the remarks of the previous sections apply.

2.3. New language features

Let us first discuss the addition of data structures of Pascal omitted in *LS*. Types *real* and *char* do not cause any problems at all. They merely require changes in the definition of *Ty*, the domain of types. Reals require some

attention in the treatment of coercion operations but a solution is straightforward. Similarly, a simple minded implementation of variant records is easy. To model the (undesirable) semantics of Pascal we simply define the location corresponding to records to have several identifiers mapping to the same location. Even simpler is the implementation that assigns disjoint storage to all variants.

Packed data structures pose an interesting problem. Specifying an object to be packed does not change the semantics in any way. The **packed** attribute is of pragmatic rather than semantic nature. Since a denotational definition is purely extensional an implementation ignoring the packed attribute is perfectly valid. Conversely, a compiler could pack all data objects and would still be correct. It is unclear how attributes like **packed** can be captured by a formal definition or whether they are desirable at all.

We omitted *for* loops, *case* and *with* statements in *LS*. Including any of these poses no problem at all. The *case* statement and the for loop could both be implemented merely by changing the tree transformations to convert it into nested conditionals and while loops respectively. A simple implementation of the with statement would require to open a new scope in which identifiers denoting record components are declared as variables. In the target language storage would be set up for these variables and their values (addresses) determined on entry to the while statement. In either case a correctness proof is immediate.

Another restriction in *LS* is that mutual recursion is disallowd. As we mentioned earlier, the code generation translates mutually recursive procedures and functions correctly. The restriction lies in the static semantics part. An extension to cover mutual recursion could be implemented in either of two ways. First, we could have forward declarations as in Pascal. This merely extends the syntax and the tree transformation part of the definition. Changes to the static semantic definition are minor. Alternatively, mutually recursive procedures could be treated similarly to recursive types, i.e. we could allow a procedure or function identifier to occur before its declaration. This requires that in the static semantic definition a fixed point is used to describe the meaning of a set of declarations.

2.4. A stronger correctness statement

We proved that whenever the compiler terminates without printing an error message then the produced code is correct. We argued earlier that is a useful statement since it will never deceive the user of the compiler to believe

that his program is correctly translated when it is not. The reader may ask how difficult it is to prove a stronger statement.

Let us first consider termination. There are well known methods to prove termination in a weak programming logic by adding counters to the program which decrease with every loop and recursion and for which one proves that they always remain positive [LS75, MP73]. It poses no particular problem to include such counters. However, additional program documentation is necessary to prove that they remain positive. For example, consider the implementation of the static semantics. To prove that the recursive procedures terminate requires to prove that the list structure representing the abstract syntax is free of cycles. This is not trivial and requires additional proofs of the tree transformation program to show that only cycle free list structures are produced.

Another possible extension is to prove that whenever the compiler prints an error message, then the input program is wrong. This is true for most error messages. For example, the static semantics defines exactly, when an error is called for. Similarly, from the LR-theory it is easily provable that an error message in the parser is only printed if the input program is in error. But at several places in the compiler error messages are printed in situations which are never expected to arise. For example, in the static semantics of expressions we consider all cases of expressions and if none of these cases obtains, an error message is printed. If the abstract syntax tree is build correctly, this situation will never arise and this error message will never occur. However, to prove that the abstract syntax is well formed is not immediate.

In some situations it will be impossible to guarantee, that no errors occur. For example, the stacks used in the parsing algorithm all have finite length But for any fixed size of these stacks there will be an input program requiring a greater size.

Providing error messages for situations which should never occur or which indicate overflow conditions is standard practice in programming. Our approach shows that these "redundancies" have their place in the framework of verification as well.

3. Future research

This thesis raises several open questions and points to research areas to improve verification. In this section we discuss some of the important issues; but clearly, the list is open ended.

3.1. Structuring a compiler

Did we choose the best formal definition? Is our semantic definition the best suited for a compiler proof? Should the structure of the compiler be different? A definite answer to tnese questions can only be given if we have other verified compilers and can compare different approaches. But some minor points should be noted here.

In the static semantics we computed the modes of all expressions to check the program for validity. This information has been discarded afterwards. Later, the code generation also requires information about modes of expressions; our compiler recomputes modes if necessary. But clearly, this is redundant as one could store the mode information in the abstract syntax tree. The main reason why we choose not to do so is a technical one. Changing the abstract syntax tree is an update operation on a pointer structure which has no obvious counterpart in the formal definition. Consequently it is not immediately clear how the correctness of such update operations can be stated.

Alternatively, static and dynamic semantics could be combined in one program. This also eliminates redundant computation of modes. In return the structure of the resulting compiler is less clean.

The idea of combining different parts into one can be carried a step further. The scanner and the parser could be combined into one program thus eliminating the representation of the program as sequence of tokens. A natural way towards this implementation are coroutines which unfortunately are not available in our implementation language. Any two programs that communicate by writing and reading one file can be combined using coroutines without any change in the program proof.

The size of programs that can be translated by our compiler is severely limited by the fact that the complete abstract syntax tree is stored in memory. But this is not strictly necessary. The semantics of LS is such that semantic ckecking and code generation can proceed from left to right requiring only a limited context. A suitable reorganisation of our program seems possible without significant changes in the formal theory. Again, here is a tradeoff between efficiency and structural clarity.

3.2. Improvements of verification systems

The research presented here may well be the first real test of any automatic verification system. Several weak points that call for improvement have been revealed.

Clearly, the assertion language used in a verifier is a compromise between

expressive power and efficiency of the system's theorem prover. For example, allowing quantification in assertions will make theorem proving much harder and may render the whole system useless.

Still it seems desirable to allow for limited quantification. Can a proof rule corresponding to our weak $\forall$-introduction be included. Even if the theorem prover does not deal with quantification, can quantified formulas be allowed syntactically. For example, one might might want to write $\forall x.P(x,y)$ and have the theorem prover treat the formula as a predicate symbol with one free variable y. Of course, this would not change anything in the capabilities of the system except for the readability of the documentation.

Another desirable feature is a typed assertion and rule language. Many proofs given here were not forthcoming because of a misspelling of a predicate or function symbol. Simple minded type checking could detect many of these errors.

The verifier should provide for virtual code. Not merely virtual variables but rather virtual types, virtual parameters, virtual procedures and so on. Such a concept is imperative if the verified program is ever to be compiled. Also, it greatly enhances the readability of programs and clarifies which parts of the code require further refinement.

The theorem proving part of the Stanford verifier performed extremely well in most cases. In particular the built-in complete decision procedures for frequently occuring theories proved to be a valuable asset. The main problems are in the area of rules supplied to the prover.

The semantic contents of a rule should be stated independently of a particular heuristics of how to use this rule. Currently, there are several different ways of stating the same fact. To get the prover to simplify a verification condition efficiently, rules have to be stated in the "right" format. This requires some experimenting and frequent rewriting of rules, a possible source of errors.

In some cases it is desirable to manually intervene in a proof and give hints to the system. Or, one may want to find out why exactly a verification condition does not simplify to true. But manual proofs should be the exception rather than the rule. Giving manual proof guidance to the system in all cases would make this research impossible. Rather what is needed is a general concepts of a "proof schema", a way of telling the system how to prove a verification condition. Consider the static semantics for expressions. The program consists of several branches each of which checks a particular syntactic class of expressions such as identifiers, unary operators, binary operators and so on. The structure of the proof for each of these cases is very similar. Therefore, given the proof of one case, all the other cases follow by analogy.

The concept of a proof schema seems very useful here.

We have shown that refinement of a program leads to structurally similar verification conditions. An intelligent verification system might remember previous proofs, thus greatly reducing the amount of work necessary to verify a refined program.

3.3. Better verification techniques

Weak ∀-introduction proved a useful concept in this research. Is it generally applicable? Is it a special concepts or can it be generalized? For example, is there weak ∃-introduction and so on?

We used operationalization of fixed points to compute recursive types. Is this just a gimmick to circumvent limitations of the verifiers assertion language or is it a generally applicable technique. What is the most general situation in which operationalization can be employed?

Even more interesting is the opposite question. Clearly, any update operation on pointers can be expressed as a least fixed point. Is this a practical way to reason about pointer operations? It certainly is, if pointers are used to represent recursive domains and if the pointer operations have a natural counterpart (e.g. fixed point) in the underlying theory. But can it also be used to prove a tree traversal algorithm which manipulates link fields to avoid stacking operations?

Alternatively, it may be possible to develop high level concepts to reason about pointers similar to those proposed by Reynolds for arrays [Re79].

3.4. Program development systems

The ideas put forth sofar all assume that the current paradigm of program verification is the best possible. Maybe, it is not. What would the ideal verification system for our compiler proof look like? Ideally, of course, we want a program synthesis system, but let us be a little more realistic.

One of the main problems in carrying out the development of the compiler and its proof was consistency. We had to deal with an extremely large formal definition, theorems derived from this definition, machine readable versions of specifications and theorems, several versions (of different refinement level) of the program to be written, and programs communication with a given program. A system that keeps track of all these formal objects and insures notational consistency seems easy to conceive and would be of great assistence.

But, such a system could do even more. We argued previously, that many proofs of cases of our program follow by analogy. But the implementation of

different clauses of the definition follow by analogy also. Thus, what is needed is an intelligent editor. For instance, the user should be able to explain to such a system, how a recursively defined function is to be transformed into an efficient program. Part of such a transformation is the specification of implementation details, for instance, how a particular abstract object is to be represented in the program. An even more advanced system might know about efficient default implementation of certain objects (automatic data structure selection).

References

Abbreviations:

- AIM — Artificial Intelligence Memo, Stanford University
- TCS — Theoretical Computer Science
- LNCS — Lecture Notes in Computer Science, Springer Verlag
- CACM — Communications of the ACM
- JACM — Journal of the ACM

[AU72] **Aho, A. V., Ullman J. D.**: *Theory of Parsing, Translation, and Compiling, vol 1,2*; Prentice Hall, Englewood Cliffs, 1972

[AJ74] **Aho, A. V., Johnson S. C.**: *LR Parsing*; Comp. Surveys, Vol. 6, No. 2 (1974)

[AA74] **Aiello, L., Aiello, M., Weyhrauch, R.**: *The Semantics of Pascal in LCF*; AIM-221, 1974

[AA77] **Aiello, L., Aiello, M., Weyhrauch R.W.**: *Pascal in LCF: Semantics and examples of proof.*; TCS 5, 1977, 135-177.

[AE73] **Anderson, T., Eve, J., Horning, J.J.**: *Efficient LR(1) Parsers*; Acta Informatica 2 (1973)

[BL70] **Birkhoff, G., Lipson, J. D.**: *Heterogeneous Algebras*; J. Comb. Theory 8, 115-133 , 1970

[Bo76] **Bochman, G.**: *Semantic evaluation from left to right*; CACM 19/2 (1976)

[BM77] **Boyer, R.S., Moore, J.S.**: *A computer proof of the correctness of a simple optimizing compiler for expressions*; SRI Tech. Report 5, 1977

[Bj77] **Bjørner, D.**: *Formal development of interpreters and compilers*; Report ID673, Technical University of Denmark, Lyngby 1977

[BJ78] **Bjørner, D., Jones, C.B.(ed.)**: *The Vienna Development Method: The Meta Language*; LNCS 61 (1978)

[Bu69] **Burstall, R. M.**: *Proving Properties of Programs by Structural Induction*; Computer J. 12/1, 1969

[BL69] **Burstall, R. M., Landin, P. J.**: *Programs and their Proofs: An Algebraic Aproach*; Machine Intelligence 4, 1969

[BC71] **Burroughs Corp.**: *Burroughs B6700 Handbook, Vol 1, Hardware*; Form No. 5000276, 1971

[BC72] **Burroughs Corp.**: *Burroughs B6700 Information Processing Systems, Reference Manul*; Form No. 1058633, 1972

[BC73] **Burroughs Corp.**: *System Miscellanea*; Form No. 5000367, 1973

[Ca76] **Cartwright, R**: *Practical Formal Semantic Definition and Verification Systems*; AIM-296, December 1976

[CM79] **Cartwright, R., McCarthy, J.**: *First order programming logic*; Sixth annual ACM Symposium on Principle of Programming languages, San Antonio, 1979

[Ch76] **Chirica L.**: *Contributions to compiler correctness.*; PhD Diss. UCLA 1976

[CM75] **Chirica, C.M., Martin, D.F**: *An approach to compiler correctness*; Intn'l Conf. on reliable software, Los Angeles, 1975

[Ch51] **Church, A.**: *The Calculi of Lambda-Conversion*; Annals of Mathematical Studies 6, Princeton University Press, Princeton (1951)

[Cl79] **Clarke E. M.**: *Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems*; JACM, 26/1, 1979

[CH79] **Cohen, R., Harry, E.**: *Automatic generation of near-optimal linear-time translators for non circular attribute grammars*; Sixth annual ACM Symposium on Principle of Programming languages, San Antonio, 1979

[Co79a] **Cohn, A.**: *High level proof in LCF*; Proceedings of the Fifth Symposium on Automated Deduction, 1979

[Co79b] **Cohn, A.**: *Machine assisted proofs of recursion implementation*; PhD Thesis, University of Edinburgh, 1979

[Co65] **Cohn, P.M.**: *Universal Algebra*; Harper & Row, 1965

[Co77] **Cook, S**: *Soundness and completeness of an axiom system for program verification*; 9th Symp. on Theory of Computing, Boulder, 1977

[DL78] **Darringer, J.A., Laventhal, M.S.:** *A Study of the Use of Abstractions in Program Specification and Verification*; IBM Research Report RC7184. 1978

[DL79] **DeMillo, R.A., Lipton, R.J., Perlis, A.J.:** *Social processes of proofs of theorems and programs*; CACM 22/5, 1979

[De78] **Deransart, P.:** *Proof and Synthesis of Semantic Attributes in Compiler Definition*; IRIA, Report 333, Dec. 1978

[De71] **DeRemer, F.L.:** *Simple LR(k) grammars*; CACM 14 (1971)

[De74] **DeRemer, F.L.:** *Transformational Grammars*; in [GH74]

[De73] **DeRemer, F.L.:** *Transformational Grammars for languages and Compilers*; Technical Report 50, Univ. of Newcastle unpon Tyne, 1973

[Di76] **Dijkstra, E.W.:** *A discipline of Programming*; Prentice–Hall, Englewood Cliffs, 1976

[Do76] **Donahue, J. E.:** *Complementary Definitions of Programming language Semantics*; LNCS 42, Springer, 1976

[Do77] **Donahue, J.:** *On the semantics of "Data Type"*; TR 77-311, CS Dept. Cornell University (1977)

[DG79] **Donzdau, V., Kahn, G., Krieg–Brückner, B.:** *Formal definition of ADA*; Preliminary Draft, CII Honeywell–Bull, Oct. 1979

[En72] **Engelfriet, J.:** *A note on infinite trees*; Information Processing Letters 1, 1972

[Fl67] **Floyd, R. W.:** *Assigning Meanings to Programs*; Proceedings of Symp. in Applied Mathematics 19 (1967)

[Go78] **Goguen, J.:** *Some Ideas in Algebraic Semantics*; Proceedings of the third IBM Symposium on mathematical foundations of computer science, 1978

[GT77] **Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright J.B.:** *Initial algebra semantics and continuous algebras.*; JACM 24.1, 1977, 68-95.

[GH74] **Goos, G., Hartmanis, J.:** *Compiler Construction, an advanced course*; LNCS 21 (1974)

[Go75] **Gordon, M.:** *Operational Reasoning and denotational Semantics*; AIM-264, 1975

[GM75] **Gordon, M., Milner, R., Wadsworth, C.:** *Edinburgh LCF*; Internal report CSR-11-77, Unversity of Edinburgh

[Go79] **Gordon, M.J.C.:** *The denotational description of programming languages, an introduction*; Springer Verlag, New York, Heidelberg, Berlin, 1979

[GM79] **Greif, I., A. Meyer:** *Specifying Programming language Semantics*; Sixth annual ACM Symposium on Principle of Programming Languages, San Antonio, 1979

[Gr71] **Gries, D.:** *Compiler Construction for Digital Computers*; John Wiley, New York, 1971

[Gu75] **Guttag, J. V.:** *The Specification and Application to Programming of Abstract Data Types*; Technical Report CSRG- 59, University of Toronto, 1975

[GH76] **Guttag, J. V., Horowitz, F., Musser, D.R.:** *Abstract Data Types and Software Validation*; USC-ISI Technical Report (1976)

[Ha73] **Habermann, A.N.:** *Critical Comments on the Programming Language Pascal*; Acta Informatica 3, pp47-57 (1973)

[He75] **vonHenke, F. W.:** *On the Representation of Data Structures in LCF with Applications to Program Generation*; AIM-267, September 1975

[HL74] **von Henke F. W., Luckham D. C.:** *Automatic Program Verification III: A Methodology for Verifying Programs*; AIM-256, 1974

[Ho69] **Hoare, C. A. R.:** *An Axiomatic Basis of Computer Programming*; CACM 12, Oct, pp 576-580 (1969)

[Ho72] **Hoare, C. A. R.:** *Proofs of Correctness of Data Representation*; Acta Informatica 1/1 (1972)

[HL74] **Hoare, C. A. R., Lauer, P.E:** *Consistent and Complimentary Formal Theories of the Semantics of Programming languages*; Acta Informatica 3, pp135-154, (1974)

[HW73] **Hoare, C. A. R., Wirth, N.:** *An Aziomatic Definition of the Programming language Pascal*; Acta Informatica, 2 (1973), pp.335-355

[Ic79] **Ichbiah, J. D. et al:** *Preliminary ADA reference manual*; Sigplan Notices 14/6, 1979

[IL75] **Igarashi, S., London, R. L., Luckham, D. C.:** *Automatic Program Verification 1: Logical Basis and Its Implementation*; Acta Informatica, Vol 4, pp 145-182 (1975)

[JW76] **Jensen, K., Wirth, N.:** *PASCAL, User Manual and Report*; Springer, New York, Heidelberg, Berlin, 1976

[Jo79] **Jones C.B.:** *Constructing a theory of a data structure as an aid to program development.*; Acta Informatica 11, 1979, 119-137.

[Ka67] **Kaplan, D. M.:** *Correctness of a Compiler for Algol-like Programs*; AIM - 48, Stanford University, 1967

[Ka76a] **Kastens, U.:** *Systematische Analyse semantischer Abhängigkeiten*; Informatik-Fachberichte 1, Springer Berlin Heidelberg New York, 1976

[Ka76b] **Kastens, U.:** *Ein übersetzer-erzeugendes System auf der Basis attributierter Grammatiken*; Dissertation, Universität Karlsruhe, 1976

[Kl67] **Kleene, S.C.:** *Mathematical Logic*; John Wiley, 1967

[Kn65] **Knuth, D.E.:** *On the translation of languages from left to right*; Information and Control 8/6, 1965

[Kn68] **Knuth, D. E.:** *Semantics of Context-Free languages*; Math. Systems Theory, vol. 2, pp. 127-145, 1968

[Kn78] **Knuth, D. E.:** *Tau Epsilon Chi, A System for Technical Text*; AIM-317, September 1978

[Kr74] **Kron, H.H.:** *Practical Subtree Transformational Grammars*; University of California at Santa Cruz, MS Thesis, 1974

[Kr75] **Kron, H.H.:** *Tree Templates and Subtree Transformational Grammars*; PhD Thesis, University of California at Santa Cruz, 1975

[Lo71] **London, R. L.:** *Correctness of two compilers for a LISP subset*; AIM - 151, Stanford University, 1971

[Lo72] **London, R. L.:** *Correctness of a compiler for a LISP subset*; Proceedings of an ACM conference on proving assertions about programs, Sigplan Notices 7/1 1972

[Lo78] **London, R.L. et al:** *Proof rules for the programming language Euclid*; Acta Informatica, Vol 10, No 1 (1978)

[LW69] **Lucas, P., Walk, K.** *On the formal description of PL/1*; Annual Review in Automatic Programming 6, 3 (1969)

[LS75] **Luckham, D. C., Suzuki, N** *Automatic Program Verification IV: Proof of Termination Within a Weak Logic of Programs*; AIM-269, October 1975

[LS76] **Luckham, D. C., Suzuki, N** *Automatic Program Verification V: Verification-Oriented Proof Rules for Arrays, Records and Pointers*; AIM-278, March 1976

[Ly78] **Lynn, D. S.:** *Interactive Compiler Proving using Hoare proof rules*; ISI/RR-78-70, 1978

[MP73] **Manna, Z., Pnueli, A.:** *Aziomatic Approach to Total Correctness of Programs*; AIM-210, July 1973

[Ma74] **Manna Z.:** *Mathematical Theory of Computation*; McGraw-Hill, New-York 1974

[Ma71] **Mazurkiewicz, A.:** *Proving Algorithms by Tail Functions*; Information and Control, 18 (1971), pp220-226

[Mc62] **McCarthy J.:** *Towards a mathematical theory of computation.*; Proceedings ICIP 1962

[Mc63] **McCarthy, J.:** *A basis for a mathematical theory of computation*; in P.Braffort and D. Hirschberg (eds.), Computer Programming and Formal systems, pp. 33-70, North-Holland, Amsterdam, 1963

[Mc66] **McCarthy, J.:** *A formal description of a subset of Algol*; Formal language description languages for computer programming (Steel, T.B., ed.), North Holland (1966)

[MP66] **McCarthy, J., Painter, J.:** *Correctness of a Compiler for Arithmetic Expressions*; AIM-40, Stanford University 1966

[Mc65] **McKeeman, W. M.**: *Peephole Optimization*; CACM 8/7 (1965)

[Me64] **Mendelson, E.**: *Introduction to mathematical logic*; van Nostrand, 1964

[MG79] **Meyer, A.R., Greif, I.**: *Can partial correctness assertions specify programming language semantics?*; LNCS 67 (1979)

[MS76] **Milne, R., Strachey, C.**: *A theory of programming language semantics*; Chapman and Hall, London 1976

[Mi72a] **Milner R.**: *Implementation and applications of Scott's Logic for Computable Functions.*; in: Proceedings of the ACM Conference on Proving Assertions about Programs, Las Cruces, 1972.

[Mi72b] **Milner, R.**: *Logic for Computable Functions: Description of a Machine Implementation*; AIM-169, May 1972

[Mi72c] **Milne, R.E.**: *The mathematical semantics of Algol 68 (manuscript)*; Programming Research Group, University of Oxford (1972)

[MW72] **Milner, R., Weyhrauch, R.**: *Proving compiler correctness in a mechanized logic*; Machine Intelligence 7, 1972

[Mo72] **Morris, F. L**: *Correctness of Translation of Programming Languages, an algebraic approach*; AIM 174, 1972

[Mo73] **Morris, F.L.**: *Advice on structuring compilers and proving them correct*; Proc. ACM symp. on Principles of Programming Languages, Boston, 1973

[Mo75a] **Mosses, P.D.**: *The Mathematical Semantics of Algol 60*; Tech. Monograph PRG-12, Programming Research Group, University of Oxford (1975)

[Mo75b] **Mosses, P.D.**: *Mathematical Semantics and Compiler Generation*; PhD thesis, University of Oxford (1975)

[NO78] **Nelson, G., Oppen, D. C.**: *Simplification by Cooperating Decision Prodcedures*; AIM-311, April 1978

[Ne73] **Newey, M.**: *Axioms and Theorems for Integers, Lists and Finite Sets in LCF*; AIM-184, January 1973

[Or73] **Organic, E.I.**: *Computer system Organization, The B5700/B6700 Series*; Academic Press, 1973

[Pa79] **Pagan, F. A.**: *Algol 68 as a metalangauge for denotational semantics*; The Computer Journal, Vol 22/1 1979

[Pa67] **Painter, J. A.**: *Semantic correctness of a compiler for an Algol - like language*; AIM - 44, Stanford University, 1967

[Pa69] **Park, D**: *Fixed point induction and proofs of programs*; Machine Intelligence 4, Edinburgh University Press, 1969

[Pl78] **Plotkin, G.**: *T^ω as a Universal Domain*; Journal of computer and system sciences 17, pp 209-236 (1978)

[Pl76] **Plotkin G.**: *A powerdomain construction.*; SIAM Journal of Computing 5, 1976, 452-487.

[Po79] **Polak, W.**: *An exercise in automatic program verification*; IEEE Transactions on Software Engineering, SE-5/5 (1979)

[Po80] **Polak, W.**: *Program verification based on denotational semantics*; forthcoming

[RR64] **Randell, Russel**: *Algol 60 Implmentation*; Academic Press, London 1964

[Re72] **Reynolds, J.C.**: *Definitional Interpreters for High-order Programming languages*; Proc. 25th ACM National Conf, Boston (1972), pp717-740

[Re74] **Reynolds, J.C.**: *On the Relation between Direct and Continuation Semantics*; Proc. 2nd Coll. on Automata, languages and Programming, Saarbrücken, pp. 157 - 168, 1974

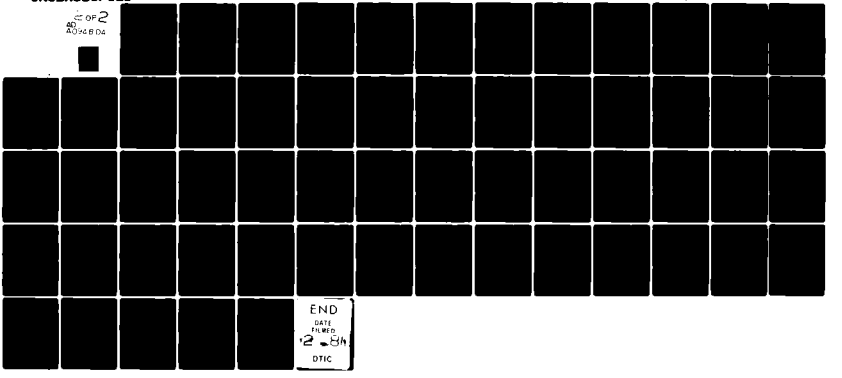
[Re79] **Reynolds J.C.**: *Reasoning about arrays.*; Communications of the ACM 22.5, 1979, 290-299.

[Ro71] **Rosen, B.K.**: *Subtree Replacement Systems*; TR 2-71, Harvard University, 1971

[Sa75] **Samet, H.**: *Automatically Proving the Correctness of Translations Involving Optimized Code*; AIM-259, May 1975

[Sc76a] **Schmeck, H.**: *Ein algebraischer Ansatz für Kompilerkorrektheitsbeweise*; (Brauer, W., ed.) Informatik - Fachberichte, Programmiersprachen, Springer, Berlin, Heidelberg, New York, 1976

AD-A094 604 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
THEORY OF COMPILER SPECIFICATION AND VERIFICATION.(U)
MAY 80 W H POLAK MDA903-76-C-0206
UNCLASSIFIED NL



[Sc78] **Schwartz, R.L.**: *An Axiomatic Semantic Definition of ALGOL 68*; CS Dept, UCLA, UCLA-34-P214-75, Aug. 78

[Sc70] **Scott, D.**: *Outline of a Mathematical Theory of Computation*; Proc. 4th annual Princeton Conf. on Information Sciences and Systems, Princeton University (1970), pp169-176

[SS71] **Scott, D., Strachey, C.**: *Toward a Mathematical Semantics for Computer languages*; Tech. Monograph PRG-6, Programming Research Group, University of Oxford (1971)

[Sc72a] **Scott D.**: *Continuous Lattices.*; Springer Lecture Notes in Mathematics, vol. 274, 97-136, 1972.

[Sc72b] **Scott, D.**: *Lattice Theory, Data Types and Semantics*; NYU Symp. on Formal Semantics, Prentice-Hall, New York (1972)

[Sc76b] **Scott, D.**: *Data Types as Lattices*; SIAM Journal of Computing, 5 (1976), pp522-587

[Sm78] **Smyth M.B.**: *Power domains.*; Journal of Computer and System Sciences 16, 1978, 23-36.

[SV79] **Stanford Verification Group**: *Stanford Pascal Verifier User Manual*; Staniord Verification Group Report No. 11, 1979

[St77] **Stoy, J.**: *Denotational Semantics — The Scott-Strachey Approach to language Theory*; MIT Press, Cambridge (1977)

[SW74] **Strachey, C., Wadsworth, C. P.**: *Continuations, a Mathematical Semantics for Handling Full Jumps*; Technical Monograph PRG-11, Oxford University, 1974

[Su75] **Suzuki, N.**: *Verifying Programs by Algebraic and Logical Reduction*; Proceedings of Int'l Conf on Reliable Software, IEEE, pp 473-481 (1975)

[Su76] **Suzuki, N.**: *Automatic Verification of Programs with Complex Data Structures*; AIM-279, February 1976

[Su80] **Suzuki, N.**: *Analysis of pointer rotation*; Seventh Annual ACM Symp. on Principles of Programming languages, Las Vegas (1980)

[Te76] **Tennent, R.D.**: *The Denotational Semantics of Programming Languages*; CACM, 19 (1976) pp437-453

[Te77a] **Tennent, R.D.**: *A Denotational Definition of the Programming language Pascal*; Tech. Report 77-47, Queen's University, Kingston, Ontario (1977)

[Te77b] **Tennent, R.D.**: *language design methods based on semantic principles*; Acta Informatica 8, (1977)

[TW79] **Thatcher, J. W., Wagner, E. G., Wright, J.B.**: *More advice on structuring compilers and proving them correct*; Report RC 7588, IBM Yorktown Heights, 1979

[Wi69] **van Wijngaarden, A. et al.**: *Report on the Algorithmic language Algol 68*; Numerische Mathematik, 14, pp 79-218. 1969

[Wi76] **van Wijngaarden, A. et al**: *Revised Report on the Algorithmic language Algol 68*; Springer Berlin Heidelberg New York, 1976

[We72] **Wegner, P.**: *The Vienna definition language*; Comp. Surveys, 4/1 (1972)

[WS77] **Welsh, J., Sneeringer, W.J., Hoare, C.A.R.**: *Ambiguities and Insecurities in Pascal*; Software practice and experience 7 (1977)

[WM72] **Weyhrauch, R., Milner R.**: *Program semantics and correctness in a mechanized logic*; Proceedings of the First USA-Japan Computer Conference, 1972.

[WT74] **Weyhrauch, R. W. Thomas, A. J.**: *FOL a Proof Checker for First-order Logic*; AIM-235, 1974

Appendix 1. Formal definition of *LS*

1. Micro Syntax of *LS*

1.1. Domains

$c \in Ch$	Characters
$s \in Str = Ch^*$	Strings
Tm	Terminal symbols
$Vl = N + \{void\}$	Token values
$Tk = Tm \times Vl$	Tokens
$h \in H = (L_{id} \to N) \times (N \to \{used, unused\})$	Identifier Tables

We use N to encode token values. Operators are assigned a unique number. Identifiers are encoded as numbers. Numerals are represented by the number they denote.

1.2. Languages L_i

$$
\begin{aligned}
le &= \{\text{"a"}, \text{"b"}, \text{"c"}, \ldots, \text{"x"}, \text{"y"}, \text{"z"}\} \\
di &= \{\text{"1"}, \text{"2"}, \ldots, \text{"9"}, \text{"0"}\} \\
dl &= \{\text{"+"}, \text{"-"}, \text{"*"}, \text{"("}, \text{")"}, \text{"["}, \text{"]"}, \text{"/"}, \\
&\quad \text{"<"}, \text{">"}, \text{"="}, \text{"-"}, \text{"\uparrow"}, \text{";"}, \text{","}\} \\
pe &= \{\text{"."}\} \\
co &= \{\text{":"}\}
\end{aligned}
$$

$$
\begin{aligned}
L_{id} &= le\ (le + di)^* \\
L_n &= di\ di^* \\
L_d &= de \\
L_p &= pe + (pe\ pe) \\
L_c &= co + (co\ \text{"="})
\end{aligned}
$$

1.3. Auxiliary definitions

$$\boxed{\kappa \in K = L_{id} \to (Tk + \{?\})}$$

$$
\begin{aligned}
\kappa\ [\![\text{array}]\!] &= \langle arraysymbol, void\rangle \\
\kappa\ [\![\text{begin}]\!] &= \langle beginsymbol, void\rangle \\
\kappa\ [\![\text{const}]\!] &= \langle constsymbol, void\rangle \\
\kappa\ [\![\text{do}]\!] &= \langle dosymbol, void\rangle \\
\kappa\ [\![\text{else}]\!] &\tilde{<} \langle elsesymbol, void\rangle \\
\kappa\ [\![\text{end}]\!] &= \langle endsymbol, void\rangle \\
\kappa\ [\![\text{fi}]\!] &= \langle fisymbol, void\rangle \\
\kappa\ [\![\text{function}]\!] &= \langle functionsymbol, void\rangle \\
\kappa\ [\![\text{goto}]\!] &= \langle gotosymbol, void\rangle \\
\kappa\ [\![\text{if}]\!] &= \langle ifsymbol, void\rangle \\
\kappa\ [\![\text{od}]\!] &= \langle odsymbol, void\rangle \\
\kappa\ [\![\text{of}]\!] &= \langle ofsymbol, void\rangle \\
\kappa\ [\![\text{procedure}]\!] &= \langle proceduresymbol, void\rangle \\
\kappa\ [\![\text{program}]\!] &= \langle programsymbol, void\rangle \\
\kappa\ [\![\text{record}]\!] &= \langle recordsymbol, void\rangle \\
\kappa\ [\![\text{repeat}]\!] &= \langle repeatsymbol, void\rangle \\
\kappa\ [\![\text{then}]\!] &= \langle thensymbol, void\rangle \\
\kappa\ [\![\text{type}]\!] &= \langle typesymbol, void\rangle \\
\kappa\ [\![\text{until}]\!] &= \langle untilsymbol, void\rangle \\
\kappa\ [\![\text{var}]\!] &= \langle varsymbol, void\rangle \\
\kappa\ [\![\text{while}]\!] &= \langle whilesymbol, void\rangle \\
\kappa\ z &= ? \text{ for all other cases}
\end{aligned}
$$

$$\boxed{newn \in H \to (H \times N)}$$

$$
\begin{aligned}
newn\ h &= \langle h[used/n], n\rangle \\
&\quad \text{where } h\ n = unused
\end{aligned}
$$

$$\boxed{\mathcal{N} \in L_n \to N}$$

$$\begin{aligned}\mathcal{N}\ \langle\rangle &= 0\\ \mathcal{N}\ d{\cdot}\langle d_0\rangle &= \text{if } d_0 = \text{"0" then } 0 + 10^*(\mathcal{N}\ d) \text{ else}\\ &\cdots\\ &\text{if } d_0 = \text{"8" then } 8 + 10^*(\mathcal{N}\ d) \text{ else } 9 + 10^*(\mathcal{N}\ d)\end{aligned}$$

1.4. Semantic Functions

$$\boxed{S_{id} \in H \to L_{id} \to (Tk \times H)}$$

$$\begin{aligned}S_{id}\ h\ s = \text{if } \kappa\ s \neq ? \text{ then } &(\kappa\ s, h) \text{ else}\\ \text{if } h\ s \neq ? \text{ then } &(h\ s, h) \text{ else } \langle\langle idsymbol, n\rangle, \hat{h}\rangle\\ \text{where } &\langle\hat{h}, n\rangle = newn\ h\\ \text{where } &\hat{h} = \hat{h}[n/s]\end{aligned}$$

$$\boxed{S_n \in H \to L_n \to (Tk \times H)}$$

$$S_n\ h\ s = \langle\langle numbersymbol, \mathcal{N}\ s\rangle, h\rangle$$

$$\boxed{S_d \in H \to L_d \to (Tk \times H)}$$

$$\begin{aligned}S_d\ h\ s = \text{if } s = \text{"+" then } &\langle\langle op4symbol, 1\rangle, h\rangle \text{ else}\\ \text{if } s = \text{"—" then } &\langle\langle op4symbol, 2\rangle, h\rangle \text{ else}\\ \text{if } s = \text{"*" then } &\langle\langle op5symbol, 3\rangle, h\rangle \text{ else}\\ &\cdots\end{aligned}$$

Operator symbols are classified as *op1symbol*, *op2symbol*, etc. according to their precedence. The exact operator is encoded as the token value.

$$\boxed{S_p \in H \to L_p \to (Tk \times H)}$$

$$\begin{aligned}S_p\ h\ s = \text{if } s = \text{"." then } &\langle\langle periodsymbol, void\rangle, h\rangle \text{ else}\\ &\langle\langle dperiodsymbol, void\rangle, h\rangle\end{aligned}$$

$$\boxed{S_c \in H \to L_c \to (Tk \times H)}$$

$$\begin{aligned}S_c\ h\ s = \text{if } s = \text{":" then } &\langle\langle colonsymbol, void\rangle, h\rangle \text{ else}\\ &\langle\langle becomessymbol, void\rangle, h\rangle\end{aligned}$$

$$\boxed{scan \in Str \to Tk^*}$$

$$scan = \Psi\ h_0.$$

$$\begin{aligned}h_0\ i = \text{if } i = \text{"read" then 1 else}\\ \text{if } i = \text{"write" then 2 else}\\ \text{if } i = \text{"eof" then 3 else}\\ \text{if } i = \text{"integer" then 4 else}\\ \text{if } i = \text{"boolean" then 5 else}\\ \text{if } i = \text{"true" then 6 else}\\ \text{if } i = \text{"false" then 7 else } \bot\\ h_0^{\#2}\ n = \text{if } 1 \leq n \leq 7 \text{ then } used \text{ else } unused\end{aligned}$$

$$\boxed{\Psi \in H \to Str \to Tk^*}$$

$$\begin{aligned}\Psi\ h\ s = \text{if } s = \langle\rangle \text{ then } &\langle\rangle \text{ else}\\ \text{if } hd\ s \in \bigcup a_i \text{ then } &\Phi\ h\ (hd\ s)\ (tl\ s) \text{ else}\\ &\Psi\ h\ (tl\ s)\\ \text{where } a_i = &\{u \mid \exists v.u{\cdot}v \in L_i, u \neq \langle\rangle\}\end{aligned}$$

$$\boxed{\Phi \in H \to Str \to Str \to Tk^*}$$

$$\begin{aligned}\Phi\ h\ s_1\ s_2 = \text{if } s_2 = \langle\rangle \text{ then } &\text{ if } s_1 \in L_i \text{ then } (S_i\ h\ s_1)^{\#1} \text{ else } error \text{ else}\\ \text{if } s_1{\cdot}\langle hd\ s_2\rangle \in \bigcup a_i \text{ then } &\Phi\ h\ (s_1{\cdot}\langle hd\ s_2\rangle)\ (tl\ s_2) \text{ else}\\ \text{if } s_1 \in L_i \text{ then } &(S_i\ h\ s_1)^{\#1}{\cdot}(\Psi\ (S_i\ h\ s_1)^{\#2}\ s2) \text{ else } error\end{aligned}$$

2. Syntax of *LS*

The following SLR-grammar defines the syntax of *LS*. All uppercase symbols are terminals, lower case symbolas are nonterminals.

[*Z_1*]	Z::= PROG eofsymbol
[*PROG_1*]	PROG::= programsymbol STMT
[*BLOCK_1*]	BLOCK::= CDEFP TDEFP VDECP PFDECP CSTMT
[*CDEFP_1*]	CDEFP::=
[*CDEFP_2*]	CDEFP::= constsymbol CDEFL
[*CDEFL_1*]	CDEFL::= CDEF semicolonsymbol
[*CDEFL_2*]	CDEFL::= CDEFL CDEF semicolonsymbol
[*CDEF_1*]	CDEF::= idsymbol equalsymbol EXPR
[*TDEFP_1*]	TDEFP::=
[*TDEFP_2*]	TDEFP::= typesymbol TDEFL
[*TDEFL_1*]	TDEFL::= TDEF semicolonsymbol
[*TDEFL_2*]	TDEFL::= TDEFL TDEF semicolonsymbol
[*TDEF_1*]	TDEF::= idsymbol equalsymbol TDEN
[*TDEN_1*]	TDEN::= lparensymbol IDL rparensymbol
[*TDEN_2*]	TDEN::= idsymbol
[*TDEN_3*]	TDEN::= idsymbol colonsymbol IDN
[*TDEN_4*]	TDEN::= numbersymbol colonsymbol IDN
[*TDEN_5*]	TDEN::= upsymbol idsymbol
[*TDEN_6*]	TDEN::= arraysymbol lbracketsymbol TDEN rbracketsymbol ofsymbol TDEN
[*TDEN_7*]	TDEN::= recordsymbol VDECL endsymbol
[*IDL_1*]	IDL::= idsymbol
[*IDL_2*]	IDL::= IDL commasymbol idsymbol
[*VDECL_1*]	VDECL::= VDEC
[*VDECL_2*]	VDECL::= VDECL semicolonsymbol VDEC
[*VDEC_1*]	VDEC::= idsymbol colonsymbol TDEN
[*VDECP_1*]	VDECP::=
[*VDECP_2*]	VDECP::= varsymbol VDECL semicolonsymbol
[*PFDECP_1*]	PFDECP::=
[*PFDECP_2*]	PFDECP::= PDEC semicolonsymbol PFDECP
[*PFDECP_3*]	PFDECP::= FDEC semicolonsymbol PFDECP
[*FDEC_1*]	FDEC::= functionsymbol idsymbol PARMS colonsymbol TDEN semicolonsymbol STMT
[*PDEC_1*]	PDEC::= proceduresymbol idsymbol PARMS semicolonsymbol STMT
[*CSTMT_1*]	CSTMT::= beginsymbol COM endsymbol
[*LST_1*]	LST::= numbersymbol colonsymbol STMT
[*LST_2*]	LST::= STMT
[*COM_1*]	COM::= LST

[*COM_2*]	COM::= COM semicolonsymbol LST
[*STMT_1*]	STMT::=
[*STMT_2*]	STMT::= VBLE becomessymbol EXPR
[*STMT_3*]	STMT::= idsymbol
[*STMT_4*]	STMT::= idsymbol lparensymbol EXPRL rparensymbol
[*STMT_5*]	STMT::= ifsymbol EXPR thensymbol COM fisymbol
[*STMT_6*]	STMT::= ifsymbol EXPR thensymbol COM elsesymbol COM fisymbol
[*STMT_7*]	STMT::= whilesymbol EXPR dosymbol COM odsymbol
[*STMT_8*]	STMT::= repeatsymbol COM untilsymbol EXPR
[*STMT_9*]	STMT::= gotosymbol numbersymbol
[*STMT_10*]	STMT::= BLOCK
[*EXPR_1*]	EXPR::= EXPR op1symbol CONJ
[*EXPR_2*]	EXPR::= CONJ
[*CONJ_1*]	CONJ::= CONJ op2symbol REL
[*CONJ_2*]	CONJ::= REL
[*CONJ_3*]	CONJ::= unopsymbol REL
[*REL_1*]	REL::= REL op3symbol SUM
[*REL_2*]	REL::= REL equalsymbol SUM
[*REL_3*]	REL::= SUM
[*SUM_1*]	SUM::= SUM op4symbol TERM
[*SUM_2*]	SUM::= TERM
[*SUM_3*]	SUM::= op4symbol TERM
[*TERM_1*]	TERM::= TERM op5symbol FACT
[*TERM_2*]	TERM::= FACT
[*FACT_1*]	FACT::= lparensymbol EXPR rparensymbol
[*FACT_2*]	FACT::= idsymbol lparensymbol EXPRL rparensymbol
[*FACT_3*]	FACT::= IDN
[*FACT_4*]	FACT::= FACT upsymbol
[*FACT_5*]	FACT::= FACT periodsymbol idsymbol
[*FACT_6*]	FACT::= FACT lbracketsymbol EXPR rbracketsymbol
[*VBLE_1*]	VBLE::= idsymbol
[*VBLE_2*]	VBLE::= VBLE upsymbol
[*VBLE_3*]	VBLE::= VBLE periodsymbol idsymbol
[*VBLE_4*]	VBLE::= VBLE lbracketsymbol EXPR rbracketsymbol
[*PARMS_1*]	PARMS::=
[*PARMS_2*]	PARMS::= lparensymbol PARML rparensymbol
[*PARML_1*]	PARML::= PARM
[*PARML_2*]	PARML::= PARML semicolonsymbol PARM
[*PARM_1*]	PARM::= varsymbol idsymbol colonsymbol idsymbol
[*PARM_2*]	PARM::= idsymbol colonsymbol idsymbol
[*EXPRL_1*]	EXPRL::= EXPR
[*EXPRL_2*]	EXPRL::= EXPRL commasymbol EXPR

[*IDN_1*] IDN::= idsymbol
[*IDN_2*] IDN::= numbersymbol

3. Abstract syntax

3.1. Syntactic Domains

$\Omega \in Op$	dyadic Operators (not further defined here)
$O \in Mop$	monadic Operators (not further defined here)
$I \in Id$	Identifiers (not further defined here)
$N \in Num$	Numerals (not further defined here)
$\Xi \in Pgm$	Programs
$E \in Exp$	Expressions
$\Theta \in Stm$	Statements
$\Gamma \in Com$	Commands
$T \in Typ$	Types
$\Delta \in Dec$	Declarations
$\Delta_c \in Cdef$	Constant Definition
$\Delta_t \in Tdef$	Type definition
$\Delta_v \in Vdec$	Variable declaration
$\Pi \in Par$	Parameters

Expressions
$E ::= I \mid O\ E \mid E_0\ \Omega\ E_1 \mid E\uparrow \mid I(E^*) \mid E_0[E_1] \mid N \mid E.I$

Commands
$\Gamma ::= N{:}\Theta \mid \Theta \mid \Gamma_0;\Gamma_1$

Statements
$\Theta ::= E_0 := E_1 \mid \mathit{if}\ E\ \mathit{then}\ \Gamma_0\ \mathit{else}\ \Gamma_1\ \mathit{fi} \mid \mathit{dummy} \mid$
$\mathit{while}\ E\ \mathit{do}\ \Gamma\ \mathit{od} \mid \mathit{repeat}\ \Gamma\ \mathit{until}\ E \mid \mathit{goto}\ N \mid I(E^*) \mid$
$\Delta^*\ \mathit{begin}\ \Gamma\ \mathit{end}$

Declarations
$\Delta ::= \mathit{const}\ \Delta_c^* \mid \mathit{type}\ \Delta_t^* \mid \mathit{var}\ \Delta_v^* \mid$
$\mathit{procedure}\ I(\Pi^*);\Theta \mid \mathit{function}\ I(\Pi^*){:}T;\Theta$

Types
$T ::= I \mid (I_1, \ldots, I_n) \mid E_1..E_2 \mid \mathit{array}[T_1]\mathit{of}\ T_2 \mid$

$\mathit{record}\ I_1{:}T_1; I_2{:}T_2; \ldots; I_n{:}T_n\ \mathit{end} \mid \uparrow I$

Constant Definitions
$\Delta_c ::= I = E$

Type Definitions
$\Delta_t ::= I = T$

Variable Declarations
$\Delta_v ::= I{:}T$

Parameters
$\Pi ::= I_1{:}I_2 \mid \mathit{var}\ I_1{:}I_2$

3.2. Constructor functions

Statements:
$\mathit{mkeif}(E, \Gamma_1, \Gamma_2) = \mathit{if}\ E\ \mathit{then}\ \Gamma_1\ \mathit{else}\ \Gamma_2\ \mathit{fi}$
$\mathit{mkeassign}(E_1, E_2) = E_1 := E_2$
$\mathit{mkewhile}(E, \Gamma) = \mathit{while}\ E\ \mathit{do}\ \Gamma\ \mathit{od}$
$\mathit{mkerepeat}(\Gamma, E) = \mathit{repeat}\ \Gamma\ \mathit{until}\ E$
$\mathit{mkegoto}(N) = \mathit{goto}\ N$
$\mathit{mkepcall}(I, (E_1, \ldots, E_n)) = I(E_1, \ldots, E_n)$
$\mathit{mkeblock}((\Delta_1, \ldots, \Delta_n), \Gamma) = \Delta_1, \ldots, \Delta_n\ \mathit{begin}\ \Gamma\ \mathit{end}$
$\mathit{mkedummy} = \mathit{empty}$

Expressions:
$\mathit{mkeexpid}(I) = I$
$\mathit{mkenumber}(N) = E$
$\mathit{mkeunop}(O, E) = OE$
$\mathit{mkebinop}(E_1, 1, E_2) = E_1 + E_2$
$\mathit{mkebinop}(E_1, 2, E_2) = E_1 - E_2$
...
$\mathit{mkederef}(E) = E\uparrow$
$\mathit{mkefcall}(I, (E_1, \ldots, E_n)) = I(E_1, \ldots, E_n)$
$\mathit{mkeindex}(E_1, E_2) = E_1[E_2]$
$\mathit{mkeselect}(E, I) = E.I$

Commands:
$\mathit{mkelabel}(N, \Theta) = N{:}\Theta$
$\mathit{mkestmt}(\Theta) = \Theta$

$mkecommandlist(\Gamma_1, \Gamma_2) = \Gamma_1; \Gamma_2$

Declarations:

$mketype((\Delta_{t1}, \ldots, \Delta_{tn})) = type\ \Delta_{t1}, \ldots, \Delta_{tn}$
$mkeconst((\Delta_{c1}, \ldots, \Delta_{cn})) = const\ \Delta_{c1}, \ldots, \Delta_{cn}$
$mkevar((\Delta_{v1}, \ldots, \Delta_{vn})) = var\ \Delta_{v1}, \ldots, \Delta_{vn}$
$mkefunction(I, (\Pi_1, \ldots, \Pi_n), T, \Theta) = function\ I(\Pi_1, \ldots, \Pi_n):T; \Theta$
$mkeprocedure(I, (\Pi_1, \ldots, \Pi_n), \Theta) = procedure\ I(\Pi_1, \ldots, \Pi_n); \Theta$

Parameters:

$mkevarp(I_1:I_2) = var\ I_1:I_2$:
$mkevalp(I_1:I_2) = I_1:I_2$:

Types:

$mkeenum((I_1, \ldots, I_n)) = (I_1, \ldots, I_n)$
$mketypeid(I) = I$
$mkearray(T_1, T_2) = array\ [T_1]\ of\ T_2$
$mkerecord(I_1:T_1, \ldots, I_n:T_n) = record\ I_1:T_1, \ldots, I_n:T_n\ end$
$mkesubrange(E_1, E_2) = E_1\ ..\ E_1$
$mkepointer(T) = \uparrow T$

Programs

$mkeprogram(\Theta) = program\ \Theta\ .$

In addition to these constructors there is a set of list constructors which are used to represent lists like $E_1, \ldots, E_n$.

mkelist	—	make a singleton list
mkeappend	—	append two lists
mkenulllist	—	make an empty list
islist	—	true for lists
isnulllist	—	true for empty lists
selfirst	—	select the first element
selrest	—	select the rest

The obvious axioms for lists hold for these functions.

We use *selrest* and *selfirst* here to indicate that these are special destructores of the abstract syntax as opposed to *hd* and *tl* which operate on arbitrary lists.

4. Tree transformations

4.1. Programs

$$\Xi(t, v) = v$$

$$\Xi(Z, 1, \tau_1, \tau_2) = \Xi(\tau_1)$$
$$\Xi(PROG, 1, \tau_1, \tau_2) = mkeprogram(\Xi(\tau_2))$$
$$\Xi(BLOCK, 1, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5) = mkeblock(mkeappend(\Xi(\tau_1), \Xi(\tau_2), \Xi(\tau_3), \Xi(\tau_4)), \Xi(\tau_5))$$

4.2. Declarations

$$\Xi(CDEFP, 1) = mkenulllist$$
$$\Xi(CDEFP, 2, \tau_1, \tau_2) = mkeconst(\Xi(\tau_2))$$
$$\Xi(CDEFL, 1, \tau_1, \tau_2) = mkelist(\Xi(\tau_1))$$
$$\Xi(CDEFL, 2, \tau_1, \tau_2, \tau_3) = mkeappend(\Xi(\tau_1), mkelist(\Xi(\tau_2)))$$
$$\Xi(CDEF, 1, \tau_1, \tau_2, \tau_3) = mkeconstdec(\Xi(\tau_1), \Xi(\tau_3))$$

$$\Xi(TDEFP, 1) = mkenulllist$$
$$\Xi(TDEFP, 2, \tau_1, \tau_2, \tau_3) = mketype(\Xi(\tau_2))$$
$$\Xi(TDEFL, 1, \tau_1, \tau_2) = mkelist(\Xi(\tau_1))$$
$$\Xi(TDEFL, 2, \tau_1, \tau_2, \tau_3) = mkeappend(\Xi(\tau_1), mkelist(\Xi(\tau_2)))$$
$$\Xi(TDEF, 1, \tau_1, \tau_2, \tau_3) = mketypedec(\Xi(\tau_1), \Xi(\tau_3))$$
$$\Xi(TDEN, 1, \tau_1, \tau_2, \tau_3) = mkeenum(\Xi(\tau_2))$$
$$\Xi(TDEN, 2, \tau_1) = mketypeid(\Xi(\tau_1))$$
$$\Xi(TDEN, 3, \tau_1, \tau_2, \tau_3) = mkesubrange(\Xi(\tau_1), \Xi(\tau_3))$$
$$\Xi(TDEN, 4, \tau_1, \tau_2, \tau_3) = mkesubrange(\Xi(\tau_1), \Xi(\tau_3))$$
$$\Xi(TDEN, 5, \tau_1, \tau_2) = mkepointer(\Xi(\tau_2))$$
$$\Xi(TDEN, 6, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7) = mkearray(\Xi(\tau_3), \Xi(\tau_6))$$
$$\Xi(TDEN, 7, \tau_1, \tau_2, \tau_3) = mkerecord(\Xi(\tau_2))$$

$$\Xi(IDL, 1, \tau_1) = mkelist(\Xi(\tau_1))$$
$$\Xi(IDL, 2, \tau_1, \tau_2, \tau_3) = mkeappend(\Xi(\tau_1), mkelist(\Xi(\tau_3)))$$
$$\Xi(VDECL, 1, \tau_1) = mkelist(\Xi(\tau_1))$$
$$\Xi(VDECL, 2, \tau_1, \tau_2, \tau_3) = mkeappend(\Xi(\tau_1), mkelist(\Xi(\tau_3)))$$
$$\Xi(VDEC, 1, \tau_1, \tau_2, \tau_3) = mkevardec(\Xi(\tau_1), \Xi(\tau_3))$$
$$\Xi(VDECP, 1) = mkenulllist$$
$$\Xi(VDECP, 2, \tau_1, \tau_2, \tau_3) = mkevar(\Xi(\tau_2))$$

$$\begin{aligned}
\mathcal{E}(\mathit{FDECP}.1) &= \mathit{mkenulllist}\\
\mathcal{E}(\mathit{PFDECP}.2,\tau_1,\tau_2,\tau_3) &= \mathit{mkeappend}(\mathit{mkelist}(\mathcal{E}(\tau_1)),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{PFDECP}.3,\tau_1,\tau_2,\tau_3) &= \mathit{mkeappend}(\mathit{mkelist}(\mathcal{E}(\tau_1)),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{FDEC}.1,\tau_1,\tau_2,\tau_3,\tau_4,\tau_5,\tau_6,\tau_7) &= \mathit{mkefunction}(\mathcal{E}(\tau_2),\mathcal{E}(\tau_3),\mathcal{E}(\tau_5),\mathcal{E}(\tau_7))\\
\mathcal{E}(\mathit{PDEC}.1,\tau_1,\tau_2,\tau_3,\tau_4,\tau_5) &= \mathit{mkeprocedure}(\mathcal{E}(\tau_2),\mathcal{E}(\tau_3),\mathcal{E}(\tau_5))
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}(\mathit{VBLE}.1,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{VBLE}.2,\tau_1,\tau_2) &= \mathit{mkederef}(\mathcal{E}(\tau_1))\\
\mathcal{E}(\mathit{VBLE}.3,\tau_1,\tau_2,\tau_3) &= \mathit{mkeselect}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{VBLE}.4,\tau_1,\tau_2,\tau_3,\tau_4) &= \mathit{mkeindex}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{ARMS}.1) &= \mathit{mkenulllist}\\
\mathcal{E}(\mathit{PARMS}.2,\tau_1,\tau_2,\tau_3) &= \mathcal{E}(\tau_2)\\
\mathcal{E}(\mathit{PARML}.1,\tau_1) &= \mathit{mkelist}(\mathcal{E}(\tau_1))\\
\mathcal{E}(\mathit{PARML}.2,\tau_1,\tau_2,\tau_3) &= \mathit{mkeappend}(\mathcal{E}(\tau_1),\mathit{mkelist}(\mathcal{E}(\tau_3)))\\
\mathcal{E}(\mathit{PARM}.1,\tau_1,\tau_2,\tau_3,\tau_4) &= \mathit{mkevarp}(\mathcal{E}(\tau_2),\mathcal{E}(\tau_4))\\
\mathcal{E}(\mathit{PARM}.2,\tau_1,\tau_2,\tau_3) &= \mathit{mkevalp}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{EXPRL}.1,\tau_1) &= \mathit{mkelist}(\mathcal{E}(\tau_1))\\
\mathcal{E}(\mathit{EXPRL}.2,\tau_1,\tau_2,\tau_3) &= \mathit{mkeappend}(\mathcal{E}(\tau_1),\mathit{mkelist}(\mathcal{E}(\tau_3)))\\
\mathcal{E}(\mathit{IDN}.1,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{IDN}.2,\tau_1) &= \mathcal{E}(\tau_1)
\end{aligned}$$

4.3. Expressions

$$\begin{aligned}
\mathcal{E}(\mathit{EXPR}.1,\tau_1,\tau_2,\tau_3) &= \mathit{mkebinop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{EXPR}.2,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{CONJ}.1,\tau_1,\tau_2,\tau_3) &= \mathit{mkebinop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{CONJ}.2,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{CONJ}.3,\tau_1,\tau_2) &= \mathit{mkeunop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2))\\
\mathcal{E}(\mathit{REL}.1,\tau_1,\tau_2,\tau_3) &= \mathit{mkebinop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{REL}.2,\tau_1,\tau_2,\tau_3) &= \mathit{mkebinop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{REL}.3,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{SUM}.1,\tau_1,\tau_2,\tau_3) &= \mathit{mkebinop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{SUM}.2,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{SUM}.3,\tau_1,\tau_2) &= \mathit{mkeunop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2))\\
\mathcal{E}(\mathit{TERM}.1,\tau_1,\tau_2,\tau_3) &= \mathit{mkebinop}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_2),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{TERM}.2,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{FACT}.1,\tau_1,\tau_2,\tau_3) &= \mathcal{E}(\tau_2)\\
\mathcal{E}(\mathit{FACT}.2,\tau_1,\tau_2,\tau_3,\tau_4) &= \mathit{mkefcall}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{FACT}.3,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{FACT}.4,\tau_1,\tau_2) &= \mathit{mkederef}(\mathcal{E}(\tau_1))\\
\mathcal{E}(\mathit{FACT}.5,\tau_1,\tau_2,\tau_3) &= \mathit{mkeselect}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{FACT}.6,\tau_1,\tau_2,\tau_3,\tau_4) &= \mathit{mkeindex}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))
\end{aligned}$$

4.4. Statements

$$\begin{aligned}
\mathcal{E}(\mathit{CSTMT}.1,\tau_1,\tau_2,\tau_3) &= \mathcal{E}(\tau_2)\\
\mathcal{E}(\mathit{LST}.1,\tau_1,\tau_2,\tau_3) &= \mathit{mkelabel}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{LST}.2,\tau_1) &= \mathit{mkestmt}(\mathcal{E}(\tau_1))\\
\mathcal{E}(\mathit{COM}.1,\tau_1) &= \mathcal{E}(\tau_1)\\
\mathcal{E}(\mathit{COM}.2,\tau_1,\tau_2,\tau_3) &= \mathit{mkecommandlist}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}(\mathit{STMT}.1) &= \mathit{mkedummy}\\
\mathcal{E}(\mathit{STMT}.2,\tau_1,\tau_2,\tau_3) &= \mathit{mkassign}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{STMT}.3,\tau_1) &= \mathit{mkepcall}(\mathcal{E}(\tau_1),\mathit{mkenulllist})\\
\mathcal{E}(\mathit{STMT}.4,\tau_1,\tau_2,\tau_3,\tau_4) &= \mathit{mkepcall}(\mathcal{E}(\tau_1),\mathcal{E}(\tau_3))\\
\mathcal{E}(\mathit{STMT}.5,\tau_1,\tau_2,\tau_3,\tau_4,\tau_5) &= \mathit{mkeif}(\mathcal{E}(\tau_2),\mathcal{E}(\tau_4),\mathit{mkestmt}(\mathit{mkedummy}))\\
\mathcal{E}(\mathit{STMT}.6,\tau_1,\tau_2,\tau_3,\tau_4,\tau_5,\tau_6,\tau_7) &= \mathit{mkeif}(\mathcal{E}(\tau_2),\mathcal{E}(\tau_4),\mathcal{E}(\tau_6))\\
\mathcal{E}(\mathit{STMT}.7,\tau_1,\tau_2,\tau_3,\tau_4,\tau_5) &= \mathit{mkewhile}(\mathcal{E}(\tau_2),\mathcal{E}(\tau_4))\\
\mathcal{E}(\mathit{STMT}.8,\tau_1,\tau_2,\tau_3,\tau_4) &= \mathit{mkerepeat}(\mathcal{E}(\tau_4),\mathcal{E}(\tau_2))\\
\mathcal{E}(\mathit{STMT}.9,\tau_1,\tau_2) &= \mathit{mkegoto}(\mathcal{E}(\tau_2))\\
\mathcal{E}(\mathit{STMT}.10,\tau_1) &= \mathcal{E}(\tau_1)
\end{aligned}$$

5. Semantics of *LS*

5.1. Semantic Domains

$T = \{TT, FF\}$	truth values
$n \in N = \{\dots, -1, 0, +1, \dots\}$	integers
$\iota \in I = \{int_{-\infty}, \dots, int_{+\infty}\}$	index values
$\nu \in Tg = \{int, bool, \nu_1, \dots, \nu_i, \dots\}$	type tags
$a \in L_r$	relative locations
$a \in L_s$	static locations
$a \in L_d$	dynamic locations
$v \in Lv_r = L_r + (Id \to Lv_r) + (I \to Lv_r)$	relative L-values
$v \in Lv_s = L_s + (Id \to Lv_s) + (I \to Lv_s)$	static L-values
$v \in Lv_d = L_d + (Id \to Lv_d) + (I \to Lv_d) + \{NIL\}$	dynamic L-values
$v \in Lv = Lv_s + Lv_d$	L-values
$f \in F = L_r \to L_s$	Frames
$\epsilon \in V = I + L_s + L_d$	values
$\sigma \in S = (L_s + L_d) \to (V + \{unused\}) \times N^* \times N^*$	stores
$A = N^* + E_R$	answers
$E_R = \{invalidindex, etc\}$	runtime errors
$C = S \to A$	dynamic continuations
$G_d = C + (V \to G)$	generalized dynamic continuations
$\gamma \in G = X \to G_d$	continuations
$\pi \in P = G \dashrightarrow G$	procedure and function values
$n \in B = N$	lexical levels
$\chi \in X = (B \to X) \times (B \to F) \times X \times B$	displays
$\rho \in U = (Id \to Lv_r) \times (Id \to P) \times (Id \to B) \times (Num \to G) \times (Num \to B) \times (L_r \to T) \times B$	environments
$\varsigma \in U_s = (Id \to Md) \times (Num \to T) \times (Tg \to T)$	static environments

5.2. Types and Modes

$\tau \in Ty$ Types
$\mu \in Md$ Modes

$$\mu = [var{:}\tau] \mid [vap{:}\tau] \mid [val{:}\tau] \mid [type{:}\tau] \mid [const{:}\iota{:}\tau] \mid [sproc] \mid [proc{:}\mu_1, \dots, \mu_n] \mid [sfun] \mid [afun{:}\mu_1, \dots, \mu_n{:}\tau] \mid [pfun{:}\mu_1, \dots, \mu_n{:}\tau]$$

$$\tau = [\nu{:}sub{:}\iota_1{:}\iota_2] \mid [\nu{:}\uparrow\tau] \mid [nil] \mid [\nu{:}array{:}\tau_1{:}\tau_2] \mid [\nu{:}record{:}\langle I_1{:}\tau_1, \dots, I_n{:}\tau_n\rangle]$$

I is considered a subset of N. Operation defined on N are used for I as well with the understanding that a range overflow results in $\perp \in I$.

6. Static Semantics of *LS*

6.1. Auxilliary functions, static Semantics

$$\boxed{distinct \in D^* \to T}$$

$$distinct\,(d_1, \dots, d_n) = \bigvee_{i,j \in \{1 \dots n\}} (i \neq j \supset d_i \neq d_j)$$

$$\boxed{typetag \in Ty \to Tg}$$

$$typetag\,[\nu{:}\dots] = \nu$$

$$\boxed{isindex \in Ty \to T}$$

$$isindex\,\tau = \tau{::}[\nu{:}sub{:}\iota_1{:}\iota_2]$$

$$\boxed{isreturnable \in Ty \to T}$$

$$isreturnable\ \tau = isindex\ \tau \vee \tau::[\nu{:}\uparrow \tau'] \vee \tau::[nil]$$

$$\boxed{overlaps, contains \in Ty \to Ty \to T}$$

$$\begin{aligned} overlaps\ [\nu{:}sub{:}\iota_1{:}\iota_2][\nu{:}sub{:}\iota_3{:}\iota_4] &= \iota_1 \le \iota_4 \wedge \iota_3 \le \iota_2 \\ contains\ [\nu{:}sub{:}\iota_1{:}\iota_2][\nu{:}sub{:}\iota_3{:}\iota_4] &= \iota_1 \le \iota_3 \wedge \iota_4 \le \iota_2 \end{aligned}$$

$$\boxed{union \in Ty \to Ty \to Ty}$$

$$union\ \tau_1\tau_2 = \text{ if } \tau_1::[\nu{:}sub{:}\iota_1{:}\iota_2] \text{ then if } \tau_2::[\nu{:}sub{:}\iota_3{:}\iota_4] \text{ then } [\nu{:}sub{:}\iota_1{:}\iota_4]$$

$$\boxed{type \in Md \to Ty}$$

$$\begin{aligned} type\ [var{:}\tau] &= \tau \\ type\ [vap{:}\tau] &= \tau \\ type\ [val{:}\tau] &= \tau \\ type\ [const{:}\epsilon{:}\tau] &= \tau \\ type\ [afun{:}\mu_1, \ldots, \mu_n{:}\tau] &= \tau \\ type\ [pfun{:}\mu_1, \ldots, \mu_n{:}\tau] &= \tau \end{aligned}$$

$$\boxed{equal \in Ty \to Ty \to T}$$

$$\begin{aligned} equal\ \tau_1\tau_2 = \text{ if } \tau_1::[\nu{:}sub{:}\iota_1{:}\iota_2] \text{ then} \\ \quad \text{if } \tau_2::[\nu{:}sub{:}\iota_1{:}\iota_2] \text{ then } true \text{ else } false \\ \text{else } typetag\ \tau_1 = typetag\tau_2 \end{aligned}$$

$$\boxed{compatible \in Ty \to Ty \to T}$$

$$\begin{aligned} compatible\ \tau_1\tau_2 = & equal\ \tau_1\tau_2 \vee overlaps\ \tau_1\tau_2 \vee \\ & (\tau_1::[\nu{:}\uparrow\tau'] \wedge \tau_2 = [nil]) \end{aligned}$$

$$\boxed{assignable \in Md \to Md \to T}$$

$$\begin{aligned} assignable\ \mu_1\mu_2 = & isvar\ \mu_1 \wedge \\ & compatible(type\ \mu_1)(type\ \mu_2) \wedge \\ & isreturnable(type\ \mu_2) \end{aligned}$$

$$\boxed{passable \in Md \to Md \to T}$$

$$\begin{aligned} passable\ \mu_1\mu_2 = \text{ if } isvar\ \mu_1 \text{ then } isvar\ \mu_2 \wedge equal(type\ \mu_1)(type\ \mu_2) \text{ else} \\ assignable\ \mu_1\ \mu_2 \end{aligned}$$

$$\boxed{isvar, isval, \in Md \to T}$$

$$\begin{aligned} isvar\ \mu &= \mu::[var{:}\tau] \vee \mu::[vap{:}\tau] \vee \mu::[afun{:}\mu_1 \ldots \mu_n{:}\tau] \\ isval\ \mu &= \mu::[val{:}\tau] \vee \mu::[const{:}\epsilon{:}\tau] \end{aligned}$$

$$\boxed{isbool, isint \in Ty \to T}$$

$$\begin{aligned} isint\ \tau &= (typetag\ \tau = int) \\ isbool\ \tau &= (typetag\ \tau = bool) \end{aligned}$$

$$\boxed{sp \in Id \to Md^* \to T}$$

$$\begin{aligned} sp[\![print]\!](\mu) &= passable\ \mu\ [val{:}[int{:}sub{:}int_{-\infty}{:}int_{+\infty}]] \\ sp[\![new]\!](\mu) &= \mu::[var{:}[\nu{:}\uparrow\tau]] \end{aligned}$$

$$\boxed{sf \in Id \to Md^* \to Md}$$

$$\begin{aligned} sf[\![eof]\!]() &= [val{:}[bool{:}sub{:}0{:}1]] \\ sf[\![read]\!]() &= [val{:}[int{:}sub{:}int_{-\infty}{:}int_{+\infty}]] \end{aligned}$$

$\boxed{truemode, falsemode \in Md}$

$$truemode = \lfloor const{:}0{:}\lceil bool{:}sub{:}0{:}0\rceil\rfloor$$
$$falsemode = \lfloor const{:}1{:}\lceil bool{:}sub{:}1{:}1\rceil\rfloor$$

$\boxed{intconst \in I \to Md}$

$$intconst\ \iota = \lfloor const{:}\iota{:}\lceil int{:}sub{:}\iota{:}\iota\rceil\rfloor$$

$\boxed{integer \in Ty}$

$$integer = \lceil int{:}sub{:}int_{-\infty}{:}int_{+\infty}\rceil$$

$\boxed{boolean \in Ty}$

$$boolean = \lceil bool{:}sub{:}0{:}1\rceil$$

$\boxed{mkevalmode \in Md \to Md}$

$$mkevalmode\ \mu = \text{ if } isval\ \mu \text{ then } \mu \text{ else } \lfloor val{:}type\ \mu\rfloor$$

$\boxed{o \in O \to Md \to Md}$

$$\begin{aligned} o[\![\neg]\!]\mu = \text{ if } & isbool\ \mu \text{ then} \\ & \text{if } \mu{::}\lfloor const{:}\iota{:}\tau\rfloor \text{ then} \\ & \quad \text{if } \iota = 0 \text{ then } falsemode \text{ else } truemode \\ & \text{else } \lfloor val{:}boolean\rfloor \\ \text{else } & \bot \end{aligned}$$

$\boxed{w \in \Omega \to Md \to Md \to Md}$

$$\begin{aligned} w[\![\wedge]\!]\mu_1\mu_2 = \text{ if } & isbool\ \mu_1 \wedge isbool\ \mu_2 \text{ then} \\ & \text{if } \mu_1{::}\lfloor const{:}\iota{:}\tau_1\rfloor \text{ then} \\ & \quad \text{if } \iota = 0 \text{ then } mkevalmode\ \mu_2 \text{ else } falsemode \\ & \text{else if } \mu_2{::}\lfloor const{:}\iota{:}\tau_2\rfloor \text{ then} \\ & \quad \text{if } \iota = 0 \text{ then } mkevalmode\ \mu_1 \text{ else } falsemode \\ & \text{else } \lfloor val{:}boolean\rfloor \\ \text{else } & \bot \end{aligned}$$

$$\begin{aligned} w[\![\vee]\!]\mu_1\mu_2 = \text{ if } & isbool\ \mu_1 \wedge isbool\ \mu_2 \text{ then} \\ & \text{if } \mu_1{::}\lfloor const{:}\iota{:}\tau_1\rfloor \text{ then} \\ & \quad \text{if } \iota = 0 \text{ then } truemode \text{ else } mkevalmode\ \mu_2 \\ & \text{else if } \mu_2{::}\lfloor const{:}\iota{:}\tau_2\rfloor \text{ then} \\ & \quad \text{if } \iota = 0 \text{ then } truemode \text{ else } mkevalmode\ \mu_1 \\ & \text{else } \lfloor val{:}boolean\rfloor \\ \text{else } & \bot \end{aligned}$$

$$\begin{aligned} w[\![+]\!]\mu_1\mu_2 = \text{ if } & isint\ \mu_1 \wedge isint\ \mu_2 \text{ then} \\ & \text{if } \mu_1{::}\lfloor const{:}\epsilon_1{:}\tau_1\rfloor \wedge \mu_2{::}\lfloor const{:}\epsilon_2{:}\tau_2\rfloor \text{ then} \\ & \quad intconst(\epsilon_1 + \epsilon_2) \\ & \text{else } \lfloor val{:}integer\rfloor \\ \text{else } & \bot \end{aligned}$$

$$\begin{aligned} w[\![-]\!]\mu_1\mu_2 = \text{ if } & isint\ \mu_1 \wedge isint\ \mu_2 \text{ then} \\ & \text{if } \mu_1{::}\lfloor const{:}\epsilon_1{:}\tau_1\rfloor \wedge \mu_2{::}\lfloor const{:}\epsilon_2{:}\tau_2\rfloor \text{ then} \\ & \quad intconst(\epsilon_1 - \epsilon_2) \\ & \text{else } \lfloor val{:}integer\rfloor \\ \text{else } & \bot \end{aligned}$$

$$\begin{aligned} w[\![*]\!]\mu_1\mu_2 = \text{ if } & isint\ \mu_1 \wedge isint\ \mu_2 \text{ then} \\ & \text{if } \mu_1{::}\lfloor const{:}\epsilon_1{:}\tau_1\rfloor \wedge \mu_2{::}\lfloor const{:}\epsilon_2{:}\tau_2\rfloor \text{ then} \\ & \quad intconst(\epsilon_1{*}\epsilon_2) \\ & \text{else } \lfloor val{:}integer\rfloor \\ \text{else } & \bot \end{aligned}$$

$$\begin{aligned} w[\![<]\!]\mu_1\mu_2 = \text{ if } & isint\ \mu_1 \wedge isint\ \mu_2 \text{ then} \\ & \text{if } \mu_1{::}\lfloor const{:}\epsilon_1{:}\tau_1\rfloor \wedge \mu_2{::}\lfloor const{:}\epsilon_2{:}\tau_2\rfloor \text{ then} \\ & \quad boolconst(\epsilon_1 < \epsilon_2) \\ & \text{else } \lfloor val{:}boolean\rfloor \\ \text{else } & \bot \end{aligned}$$

$$w[\![>]\!]\mu_1\mu_2 = \textbf{if } isint\ \mu_1 \wedge isint\ \mu_2 \textbf{ then}$$
$$\textbf{if } \mu_1{::}\lfloor const{:}\epsilon_1{:}\tau_1\rfloor \wedge \mu_2{::}\lfloor const{:}\epsilon_2{:}\tau_2\rfloor \textbf{ then}$$
$$boolconst(\epsilon_1 > \epsilon_2)$$
$$\textbf{else } \lfloor val{:}boolean\rfloor$$
$$\textbf{else } \perp$$

6.2. Declarations

$$\boxed{d \in Dec \to U_s \to U_s}$$

$$d[\![const\Delta_{c1};\ldots;\Delta_{cn}]\!]\varsigma = dc^*[\![\Delta_{c1};\ldots;\Delta_{cn}]\!]\varsigma$$
$$d[\![type\Delta_{t1};\ldots;\Delta_{tn}]\!]\varsigma = fix(dt^*[\![\Delta_{t1};\ldots;\Delta_{tn})]\!]\varsigma)$$
$$d[\![var\Delta_{v1};\ldots;\Delta_{vn}]\!]\varsigma = dv^*[\![\Delta_{v1};\ldots;\Delta_{vn})]\!]\varsigma$$
$$d[\![procedure\ I(\Pi_1,\ldots,\Pi_n);\Theta]\!]\varsigma_0 =$$
$$\textbf{let } ((\mu_1,\ldots,\mu_n),\varsigma_n) :: p^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_0 \textbf{ in}$$
$$\textbf{if } distinct(i^*[\![\Pi_1,\ldots,\Pi_n]\!]{\cdot}i[\![\Theta]\!]) \textbf{ then}$$
$$\textbf{if } g[\![\Theta]\!]\varsigma_n\big[\lfloor proc(\mu_1,\ldots,\mu_n)\rfloor/I\big] \textbf{ then}$$
$$\varsigma_0\big[\lfloor proc(\mu_1,\ldots,\mu_n)\rfloor/I\big]$$
$$d[\![function\ I_1(\Pi_1,\ldots,\Pi_n){:}I_2;\Theta]\!]\varsigma_0 =$$
$$\textbf{let } ((\mu_1,\ldots,\mu_n),\varsigma_n) :: p^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_0 \textbf{ in}$$
$$\textbf{if } \lfloor type\ \tau\rfloor :: \varsigma_n[\![I_2]\!] \textbf{ then}$$
$$\textbf{if } isreturnable(\tau) \textbf{ then}$$
$$\textbf{if } distinct(i^*[\![\Pi_1,\ldots,\Pi_n]\!]\&i[\![\Theta]\!]) \textbf{ then}$$
$$\textbf{if } g[\![\Theta]\!]\varsigma_n\big[\lfloor afun(\mu_1,\ldots,\mu_n){:}\tau\rfloor/I\big] \textbf{ then}$$
$$\varsigma_0\big[\lfloor pfun(\mu_1,\ldots,\mu_n){:}\tau\rfloor/I\big]$$

$$\boxed{dt \in Tdef \to U_s \to U_s \to U_s}$$

$$dt[\![I = T]\!]\varsigma_0\varsigma_1 = \textbf{let } (\tau,\varsigma_2) = t[\![T]\!]\varsigma_0\varsigma_1 \textbf{ in } \varsigma_2\big[\lfloor type{:}\tau\rfloor/I\big]$$

$$\boxed{dt^* \in Tdef^* \to U_s \to U_s \to U_s}$$

$$dt^*[\![]\!]\varsigma_0\varsigma_1 = \varsigma_0$$
$$dt^*[\![\Delta_{t0};\Delta_{t1};\ldots;\Delta_{tn}]\!]\varsigma_0\varsigma_1 = dt^*[\![\Delta_{t1};\ldots;\Delta_{tn}]\!](dt[\![\Delta_{t0}]\!]\varsigma_0\varsigma_1)\varsigma_1$$

$$\boxed{dc \in Cdef \to U_s \to U_s}$$

$$dc[\![I = E]\!]\varsigma = \textbf{if } \lfloor const{:}\epsilon{:}\tau\rfloor :: e[\![E]\!]\varsigma \textbf{ then } \varsigma\big[\lfloor const{:}\epsilon{:}\tau\rfloor/I\big]$$

$$\boxed{dc^* \in Cdef^* \to U_s \to U_s}$$

$$dc^*[\![]\!]\varsigma = \varsigma$$
$$dc^*[\![\Delta_{c0};\Delta_{c1};\ldots;\Delta_{cn}]\!]\varsigma = dc^*[\![\Delta_{c1};\ldots;\Delta_{cn}]\!](dc[\![\Delta_{c0}]\!]\varsigma)$$

$$\boxed{dv \in Vdec \to U_s \to U_s}$$

$$dv[\![I{:}T]\!]\varsigma = \textbf{let } (\tau,\varsigma_1) = t[\![T]\!]\varsigma\varsigma \textbf{ in } \varsigma_1\big[\lfloor var{:}\tau\rfloor/I\big]$$

$$\boxed{dv^* \in Vdec^* \to U_s \to U_s}$$

$$dv^*[\![]\!]^* = \varsigma$$
$$dv^*[\![\Delta_{v0};\Delta_{v1};\ldots;\Delta_{vn}]\!]\varsigma = dv^*[\![\Delta_{v1};\ldots;\Delta_{vn}]\!](dv[\![\Delta_{v0}]\!]\varsigma)$$

$$\boxed{p \in Par \to U_s \to (Md \times U_s)}$$

$$p[\![I_1{:}I_2]\!]\varsigma = \textbf{if } \lfloor type\ \tau\rfloor :: \varsigma[\![I_2]\!] \textbf{ then}$$
$$\textbf{if } isreturnable\ \tau \textbf{ then } (\lfloor val{:}\tau\rfloor, \varsigma\big[\lfloor var{:}\tau\rfloor/I_1\big])$$
$$p[\![var\ I_1{:}I_2]\!]\varsigma = \textbf{if } \lfloor type\ \tau\rfloor :: \varsigma[\![I_2]\!] \textbf{ then } (\lfloor var{:}\tau\rfloor, \varsigma\big[\lfloor vap{:}\tau\rfloor/I_1\big])$$

$$\boxed{p^* \in Par^* \to U_s \to (Md^* \times U_s)}$$

$$p^*[\![\pi_1, \ldots, \pi_n]\!]\varsigma_0 = \langle(\mu_1, \ldots, \mu_n), \varsigma_n\rangle \quad \textbf{where } (\mu_i, \varsigma_i) = p[\![\pi_i]\!]\varsigma_{i-1}$$

$$\boxed{d^* \in Dec^* \to U_s \to U_s}$$

$$d^*[\![()]\!]\varsigma = \varsigma$$
$$d^*[\![(\Delta_1, \ldots, \Delta_n)]\!]\varsigma = d^*[\![(\Delta_2, \ldots, \Delta_n)]\!](d[\![\Delta_1]\!]\varsigma)$$

$$\boxed{t \in Typ \to U_s \to U_s \to (Ty \times U_s)}$$

$$t[\![I]\!]\varsigma_0\varsigma_1 = \textbf{if } \lfloor type\ \tau \rfloor :: \varsigma_0[\![I]\!] \textbf{ then } \langle \tau, \varsigma_0\rangle$$

$$t[\![(I_1, \ldots, I_n)]\!]\varsigma_0\varsigma_1 = \langle \lceil \nu{:}sub{:}1{:}n \rceil, \varsigma_2[\mu_i/I_i]\rangle \quad \textbf{where } \mu_i = \lfloor const{:}i{:}\lceil \nu{:}i{:}i \rceil \rfloor \quad \textbf{where } (\nu, \varsigma_2) = newtag\ \varsigma_0$$

$$t[\![E_1 \,..\, E_2]\!]\varsigma_0\varsigma_1 = \textbf{if } \lfloor const{:}\iota_1{:}\tau_1 \rfloor :: e[\![E_1]\!]\varsigma_0 \textbf{ then if } \lfloor const{:}\iota_2{:}\tau_2 \rfloor :: e[\![E_2]\!]\varsigma_0 \textbf{ then } \langle (union\ \tau_1\ \tau_2), \varsigma_0\rangle$$

$$t[\![array[T_1]\ of\ T_2]\!]\varsigma_0\varsigma_1 = \textbf{let } \langle \tau_1, \varsigma_2\rangle = t[\![T_1]\!]\varsigma_0\varsigma_1,\ \langle \tau_2, \varsigma_3\rangle = t[\![T_2]\!]\varsigma_2\varsigma_1 \textbf{ in if } isindex\ \tau_1 \textbf{ then } \langle \lceil \nu{:}array{:}\tau_1{:}\tau_2 \rceil, \varsigma_4\rangle \quad \textbf{where } (\nu, \varsigma_4) = newtag\ \varsigma_3$$

$$t[\![record\ I_1{:}T_1; I_2{:}T_2; \ldots; I_n{:}T_n\ end]\!]\varsigma_0\varsigma = \textbf{if } distinct\langle I_1, \ldots, I_n\rangle \textbf{ then let } (\tau_i, \varsigma_1) = t[\![T_i]\!]\varsigma_{i-1}\varsigma \textbf{ in if } (\nu, \varsigma_{n+1}) = newtag\ \varsigma_n \textbf{ then } \langle \lceil \nu{:}record{:}\langle I_1{:}\tau_1, \ldots, I_n{:}\tau_n\rangle \rceil, \varsigma_{n+1}\rangle$$

$$t[\![\uparrow I]\!]\varsigma_0\varsigma_1 = \textbf{if } \lfloor type{:}\tau \rfloor :: \varsigma_1[\![I]\!] \textbf{ then } \langle \lfloor \nu{:}\uparrow\tau \rfloor, \varsigma_2\rangle \quad \textbf{where } (\nu, \varsigma_2) = newtag\ \varsigma_0$$

6.3. Expressions

$$\boxed{e \in Exp \to U_s \to Md}$$

$$e[\![N]\!]\varsigma = \lfloor const{:}n{:}\lceil int{:}sub{:}n{:}n \rceil \rfloor \quad \textbf{where } n = value[\![N]\!]$$

$$e[\![I]\!]\varsigma = \textbf{if } \varsigma[\![I]\!] :: \lfloor type{:}\tau \rfloor \textbf{ then } error \textbf{ else if } \varsigma[\![I]\!] :: \lfloor proc{:}\ldots \rfloor \textbf{ then } error \textbf{ else if } \varsigma[\![I]\!] :: \lfloor sproc \rfloor \textbf{ then } error \textbf{ else } \varsigma[\![I]\!]$$

$$e[\![O\ E]\!]\varsigma = o[\![O]\!](e[\![E]\!]\varsigma)$$

$$e[\![E_0\ \Omega\ E_1]\!]\varsigma = w[\![\Omega]\!](e[\![E_0]\!]\varsigma)(e[\![E_1]\!]\varsigma)$$

$$e[\![I(E_1, \ldots, E_n)]\!]\varsigma = \textbf{if } \varsigma[\![I]\!] :: \lfloor afun{:}\mu_1, \ldots, \mu_n{:}\tau \rfloor \textbf{ then if } passable^*((\mu_1, \ldots, \mu_n), e^*[\![E_1, \ldots, E_n]\!]\varsigma) \textbf{ then } \lfloor val{:}\tau \rfloor \textbf{ if } [\![I]\!] :: \lfloor pfun{:}\mu_1, \ldots, \mu_n{:}\tau \rfloor \textbf{ then if } passable^*(\langle \mu_1, \ldots, \mu_n\rangle, e^*[\![E_1, \ldots, E_n]\!]\varsigma) \textbf{ then } \lfloor val{:}\tau \rfloor \textbf{ if } \varsigma[\![I]\!] = \lfloor sfun \rfloor \textbf{ then } sf[\![I]\!](e^*[\![E_1, \ldots, E_n]\!]\varsigma)$$

$$e[\![E.I]\!]\varsigma = \textbf{if } type(e[\![E]\!]\varsigma) :: \lceil \nu{:}record{:}\langle \ldots, I{:}\tau, \ldots\rangle \rceil \textbf{ then if } isval\ e[\![E]\!]\varsigma \textbf{ then } \lfloor val{:}\tau \rfloor \textbf{ else if } isvar\ e[\![E]\!]\varsigma \textbf{ then } \lfloor var{:}\tau \rfloor$$

$$e[\![E_0[E_1]]\!]\varsigma = \textbf{if } \lceil \nu{:}array\ \tau_1\ of\ \tau_2 \rceil :: type\ e[\![E_0]\!]\varsigma \textbf{ then if } compatible\ \tau_1(type\ e[\![E_1]\!]\varsigma) \textbf{ then if } isvar\ e[\![E_0]\!]\varsigma \textbf{ then } \lfloor var{:}\tau_2 \rfloor \textbf{ else if } isval\ e[\![E_0]\!]\varsigma \textbf{ then } \lfloor val{:}\tau_2 \rfloor$$

$$e[\![E\uparrow]\!]\varsigma = \textbf{if } type(e[\![E]\!]\varsigma) :: \lceil \nu{:}\uparrow\tau \rceil \textbf{ then } \lfloor var{:}\tau \rfloor$$

$$\boxed{e^* \in Exp^* \to U_s \to Md^*}$$

$$e^*[\![(E_1, \ldots, E_n)]\!]\varsigma = \langle e[\![E_1]\!]\varsigma, \ldots, e[\![E_n]\!]\varsigma\rangle$$

6.4. Statements

$$\boxed{g \in Stm \to U_s \to T}$$

$$g[\![E_1 := E_2]\!]\varsigma = assignable(e[\![E_1]\!]\varsigma)(e[\![E_2]\!]\varsigma)$$

$$\begin{aligned} g[\![if\ E\ then\ \Gamma_1\ else\ \Gamma_2\ fi]\!]\varsigma = {} & isbool(type\ e[\![E]\!]\varsigma) \wedge \\ & distinct(j[\![\Gamma_1]\!]) \wedge \\ & distinct(j[\![\Gamma_2]\!]) \wedge \\ & c[\![\Gamma_1]\!]\varsigma\big[(\ldots, true, \ldots)/j[\![\Gamma_1]\!]\big] \wedge \\ & c[\![\Gamma_2]\!]\varsigma\big[(\ldots, true, \ldots)/j[\![\Gamma_2]\!]\big] \end{aligned}$$

$$g[\![dummy]\!]\varsigma = true$$

$$\begin{aligned} & g[\![while\ E\ do\ \Gamma\ od]\!]\varsigma = \\ & g[\![repeat\ \Gamma\ until\ E]\!]\varsigma = isbool(type\ e[\![E]\!]\varsigma) \wedge \\ & \qquad distinct(j[\![\Gamma]\!]) \wedge \\ & \qquad c[\![\Gamma]\!]\varsigma\big[(\ldots, true, \ldots)/j[\![\Gamma]\!]\big] \end{aligned}$$

$$g[\![goto\ N]\!]\varsigma = \varsigma[\![N]\!]$$

$$\begin{aligned} g[\![I(E_1, \ldots, E_n)]\!]\varsigma = {} & \textbf{if } \varsigma[\![I]\!] :: \lfloor proc{:}\mu_1, \ldots, \mu_n \rfloor \textbf{ then} \\ & \quad passable^*(\mu_1, \ldots, \mu_n)(e^*[\![E_1, \ldots, E_n]\!]\varsigma) \textbf{ else} \\ & \textbf{if } \varsigma[\![I]\!] :: \lfloor sproc \rfloor \textbf{ then} \\ & \quad sp[\![I]\!](e^*[\![E_1, \ldots, E_n]\!]\varsigma) \end{aligned}$$

$$\begin{aligned} g[\![\Delta_1, \ldots, \Delta_n\ begin\ \Gamma\ end]\!]\varsigma = {} & distinct(j[\![\Gamma]\!]) \wedge \\ & c[\![\Gamma]\!](d^*[\![\Delta_1, \ldots, \Delta_n]\!]\varsigma_1) \\ & \textbf{where } \varsigma_1 = \varsigma\big[(\ldots, true, \ldots)/j[\![\Gamma]\!]\big]) \end{aligned}$$

$$\boxed{j \in Com \to Num^*}$$

$$\begin{aligned} j[\![N{:}\Theta]\!] &= \langle N \rangle \\ j[\![\Gamma_1; \Gamma_2]\!] &= j[\![\Gamma_1]\!] \cdot j[\![\Gamma_2]\!] \\ j[\![\Theta]\!] &= \langle\rangle \end{aligned}$$

$$\boxed{k \in (Blo + Decl) \to Id^*}$$

$$\begin{aligned} k[\![\Delta_1, \ldots, \Delta_n; begin\ \Gamma\ end]\!] &= k^*[\![\Delta_1, \ldots, \Delta_n]\!] \cdot k[\![\Gamma]\!] \\ k[\![const\ I_1 = E_1; \ldots I_n = E_n]\!] &= \langle\rangle \\ k[\![procedure\ I \ldots]\!] &= \langle I \rangle \\ k[\![function\ I \ldots]\!] &= \langle I \rangle \\ k[\![type\ I_1 = T_1; \ldots I_n = T_n]\!] &= \langle\rangle \\ k[\![var\ I_1{:}T_1; \ldots I_n; T_n]\!] &= \langle\rangle \end{aligned}$$

$$\boxed{(Par + Stm + Dec + Typ) \to Id^*}$$

$$\begin{aligned} i[\![I_1{:}I_2]\!] &= \langle I_1 \rangle \\ i[\![var\ I_1{:}I_2]\!] &= \langle I_1 \rangle \\ i[\![\Delta_1; \ldots; \Delta_n; begin\ \Gamma\ end]\!] &= i[\![\Delta_1]\!] \cdot \ldots \cdot i[\![\Delta_n]\!] \\ i[\![const\ I_1 = E_1; \ldots I_n = E_n]\!] &= \langle I_1, \ldots, I_n \rangle \\ i[\![procedure\ I \ldots]\!] &= \langle I \rangle \\ i[\![function\ I \ldots]\!] &= \langle I \rangle \\ i[\![type\ I_1 = T_1; \ldots I_n = T_n]\!] &= \langle I_1, \ldots, I_n \rangle \cdot i[\![T_1]\!] \cdot \ldots \cdot i[\![T_n]\!] \\ i[\![var\ I_1{:}T_1; \ldots I_n; T_n]\!] &= \langle I_1, \ldots, I_n \rangle \cdot i[\![T_1]\!] \cdot \ldots \cdot i[\![T_n]\!] \\ i[\![(I_1, \ldots, I_n)]\!] &= \langle I_1, \ldots, I_n \rangle \\ i[\![array\big[T_1\big]of\ T_2]\!] &= i[\![T_1]\!] \cdot i[\![T_2]\!] \\ i[\![record\ I_1{:}T_1; \ldots I_n{:}T_n\ end]\!] &= \langle I_1, \ldots, I_n \rangle \cdot i[\![T_1]\!] \cdot \ldots \cdot i[\![T_n]\!] \\ i &= \langle\rangle \text{ for all other cases.} \end{aligned}$$

$$\boxed{c \in Com \to U_s \to T}$$

$$\begin{aligned} c[\![N{:}\Theta]\!]\varsigma &= g[\![\Theta]\!]\varsigma \\ c[\![\Gamma_1; \Gamma_2]\!]\varsigma &= c[\![\Gamma_1]\!]\varsigma \wedge c[\![\Gamma_2]\!]\varsigma \\ c[\![\Theta]\!]\varsigma &= g[\![\Theta]\!]\varsigma \end{aligned}$$

7. Dynamic Semantics of *LS*

7.1. Auxiliary functions

$$\boxed{apply \in G \to V \to G}$$

$$apply\ \gamma\epsilon = \lambda\chi.\gamma\chi\epsilon$$

$$\boxed{change \in G_d \to (S \to S) \to G_d}$$

$$change\ \gamma\ f = \lambda\chi.\ \textbf{let}\ \theta = \gamma\chi\ \textbf{in}\ \textbf{if}\ \gamma \in C\ \textbf{then}\ \lambda\sigma.\gamma(f\ \sigma)\ \textbf{else}\ \lambda\epsilon.change(\gamma\epsilon)f$$

$$\boxed{update \in G \to G}$$

$$update\ \gamma = \lambda\chi\epsilon a.change\ (\gamma\chi)(\lambda\sigma.\sigma[\epsilon/a])$$

$$\boxed{store \in (S \to G_d) \to G_d}$$

$$store\ f = \textbf{if}\ f \in (S \to C)\ \textbf{then}\ \lambda\sigma.f\sigma\sigma\ \textbf{else}\ \lambda\epsilon.store(\lambda\sigma.f\sigma\epsilon)$$

$$\boxed{content \in G \to G}$$

$$content\ \gamma = \lambda\chi a.store(\lambda\sigma.(\gamma\chi)(\sigma a))$$

$$\boxed{cond \in G \to G \to G}$$

$$cond\gamma_1\gamma_2 = \lambda\chi\epsilon.\ \textbf{if}\ (\epsilon \mid I) = 0\ \textbf{then}\ \gamma_1\chi\ \textbf{else}\ \gamma_2\chi)$$

$$\boxed{return \in Id \to U \to G \to G}$$

$$return[\![I]\!]\rho\gamma = apply(content\ gamma)(\rho^{\#1}[\![I]\!])$$

$$\boxed{binop \in Op \to G \to G}$$

$$binop[\![\Omega]\!]\gamma = \lambda\chi\epsilon_2\epsilon_1.\gamma\chi(\mathcal{W}[\![\Omega]\!]\epsilon_1\epsilon_2)$$

$$\boxed{unop \in Mop \to G \to G}$$

$$unop[\![O]\!]\gamma = \lambda\chi\epsilon_1.\gamma\chi(\mathcal{O}[\![O]\!]\epsilon_1)$$

$$\boxed{Sp \in Id \to Md^* \to G \to G}$$

$$Sp[\![print]\!](\mu)\gamma = \lambda\chi\epsilon.store(\lambda\sigma.(\gamma\chi)(\sigma^{\#1}, \sigma^{\#2}, \sigma^{\#3}\cdot\epsilon))$$

$$Sp[\![new]\!](\mu)\gamma = \lambda\chi a.\ \textbf{if}\ \lfloor var{:}\tau \rfloor :: \mu\ \textbf{then}\ change(\gamma\chi)(\lambda\sigma.\ \textbf{let}\ \langle \acute{a}, \acute{\sigma} \rangle = heapl\ \tau\sigma\ \textbf{in}\ \acute{\sigma}[\acute{a}/a])\ \textbf{else}\ \textbf{if}\ \lfloor vap{:}\tau \rfloor :: \mu\ \textbf{then}\ change(\gamma\chi)(\lambda\sigma.\ \textbf{let}\ \langle \acute{a}, \acute{\sigma} \rangle = heapl\ \tau\sigma\ \textbf{in}\ \acute{\sigma}[\acute{a}/\sigma a])$$

$$\boxed{Sf \in Id \to G \to G}$$

$$Sf[\![eof]\!]\gamma = \lambda\chi.store\epsilon(\lambda\sigma.\ \textbf{if}\ \mid \sigma^{\#2} \mid = 0\ \textbf{then}\ \gamma\ 0\ \sigma\ \textbf{else}\ \gamma\ 1\ \sigma)$$

$$Sf[\![read]\!]\gamma = \lambda\chi.store\epsilon(\lambda\sigma.\ \textbf{if}\ \sigma^{\#2} :: \epsilon\cdot w\ \textbf{then}\ \gamma\ \epsilon\ \langle \sigma^{\#1}, w, \sigma^{\#3} \rangle\ \textbf{else}\ eoferror)$$

$$\boxed{verifys \in I \to I \to G \to G}$$

$$verifys\ \iota_0\ \iota_1\ \gamma = \lambda\chi\epsilon.\ \textbf{if}\ \iota_1 \leq \epsilon \wedge \epsilon \leq \iota_2\ \textbf{then}\ \gamma\chi\epsilon\ \textbf{else}\ adjust\ invalidindex(args\ \gamma\chi\epsilon)$$

$$\boxed{verifyn \in G \to G}$$

$$verifyn\ \gamma = \lambda\chi\epsilon.\ \textbf{if}\ \epsilon = nil\ \textbf{then}\ adjust\ dereferror(args\ \gamma\chi\epsilon)\ \textbf{else}\ \gamma\chi\epsilon$$

$$\boxed{index \in G \to G}$$

$$index\ \gamma = \lambda\chi\iota\ a.\gamma\chi(a\ \iota)$$

$$\boxed{select \in Id \to G \to G}$$

$$select[\![I]\!]\gamma = \lambda\chi a.\gamma\chi(a[\![I]\!])$$

$$\boxed{args \in G_d \to N}$$

$$args\ \gamma = \textbf{if}\ \gamma \in C\ \textbf{then}\ 0\ \textbf{else}\ 1 + args(\gamma\perp)$$

$$\boxed{adjust \in G_d \to N \to G_d}$$

$$adjust\ \gamma\ n = \textbf{if}\ n = 0\ \textbf{then}\ \gamma\ \textbf{else}\ adjust(\lambda\epsilon.\gamma)(n-1)$$

$$\boxed{new \in U \to (L_r \times U)}$$

$$new\ \rho = (a, \rho[true/a])$$
for an a such that $\rho a = false$.

We do not further specify *new*; any continuous function satisfying the above condition may be substituted.

$$\boxed{newl \in Ty \to U \to (Lv_r \times U)}$$

$$newl\ \tau\ \rho = \textbf{if}\ \tau::[\nu_1{:}array{:}[\nu_2{:}sub{:}\iota_1{:}\iota_2]{:}\tau]\ \textbf{then}\ newa\ \iota_1\ \iota_2\ \tau\ \rho$$
$$\textbf{else if}\ \tau::[\nu{:}record{:}(I_1{:}\tau_1, \ldots, I_n{:}\tau_n)]\ \textbf{then}\ (a, \rho_n)$$
$$\textbf{where}\ a = \perp[a_1/I_1, \ldots, a_n/I_n],$$
$$(a_i, \rho_i) = newl\ \tau_i\ \rho_{i-1}, \rho_0 = \rho$$
$$\textbf{else}\ new\ \rho$$

$$\boxed{newa \in I \to I \to Ty \to U \to (Lv_r \times U)}$$

$$newa\ \iota_1\ \iota_2\ \tau\ \rho = \textbf{if}\ \iota_1 = \iota_2\ \textbf{then}\ (\perp[a/\iota_1], \hat{\rho})$$
$$\textbf{where}\ (a, \hat{\rho}) = newl\ \tau\ \rho$$
$$\textbf{else}\ (a_1[a_2/\iota_1], \rho_2)$$
$$\textbf{where}\ (a_1, \rho_1) = newa(\iota_1 + 1)\iota_2\ \tau\ \rho$$
$$(a_2, \rho_2) = newl\ \tau\ \rho_1$$

$$\boxed{cleara \in I \to I \to Ty \to Lv \to G \to G}$$

$$cleara\ \iota_1\ \iota_2\ \tau\ a = \textbf{if}\ \iota_1 = \iota_2\ \textbf{then}\ clear\ \tau(a\ \iota_1)\ \textbf{else}$$
$$\lambda\ \gamma.clear\ \tau(a\ \iota_1); cleara(\iota_1 + 1)\iota_2\ \tau\ a\ \gamma$$

$$\boxed{clear \in Ty \to Lv \to G \to G}$$

$$clear\ \tau\ a\ \gamma = \textbf{if}\ \tau::[\nu_1{:}array{:}[\nu_2{:}sub{:}\iota_1{:}\iota_2]{:}\tau]$$
$$\textbf{then}\ cleara\ \iota_1\ \iota_2\ \tau\ a\ \gamma$$
$$\textbf{else if}\ \tau::[\nu{:}record{:}(I_1{:}\tau_1, \ldots, I_n{:}\tau_n)]$$
$$\textbf{then}\ clear\ \tau_1(a[\![I_1]\!]); \ldots clear\ \tau_n(a[\![I_n]\!]); \gamma$$
$$\textbf{else}\ (update\ \gamma)0\ a$$

$$\boxed{heap \in S \to (Lv_d \times S)}$$

$$heap\ \sigma = (a, \sigma[0/a])$$
for an a such that $\sigma a = unused$.

$$\boxed{heapa \in I \to I \to Ty \to S \to (Lv_d \times S)}$$

$$\begin{aligned} heapa\ \iota_1\ \iota_2\ \tau\ \sigma = \textbf{if } \iota_1 = \iota_2 \textbf{ then } & (\perp[a/\iota_1], \acute{\sigma}) \\ & \textbf{where } (a, \acute{\sigma}) = heapl\ \tau\ \sigma \\ \textbf{else } & (a_1[a_2/\iota_1], \sigma_2) \\ & \textbf{where } (a_1, \sigma_1) = heapa(\iota_1 + 1)\iota_2\ \tau\ \sigma \\ & (a_2, \sigma_2) = heapl\ \tau\ \sigma_1 \end{aligned}$$

$$\boxed{heapl \in Ty \to S \to (Lv_v \times S)}$$

$$\begin{aligned} heapl\ \tau\ \sigma = \textbf{if } & \tau :: [\nu_1 : array : [\nu_2 : sub : \iota_1 : \iota_2] : \acute{\tau}] \\ & \textbf{then } heapa\ \iota_1\ \iota_2\ \acute{\tau}\ \sigma \\ \textbf{else } & \textbf{if } \tau :: [\nu : record : (I_1 : \tau_1, \ldots, I_n : \tau_n)] \textbf{ then } (a, \sigma_n) \\ & \textbf{where } a = \perp[a_1/I_1, \ldots, a_n/I_n], \\ & (a_i, \sigma_i) = heapl\ \tau_i\ \sigma_{i-1}, \sigma_0 = \sigma \\ \textbf{else } & heap\ \sigma \end{aligned}$$

$$\boxed{enter \in B \to G \to G}$$

$$enter\ n\ \gamma = \lambda\chi.\gamma(up_S\chi\ n)$$

$$\boxed{up_S \in X \to B \to X}$$

$$\begin{aligned} up_S\chi\ n = \ & \textbf{fix } \lambda\acute{\chi}.(x, y, \chi, n) \\ & \textbf{where } x = \lambda\ m.\ \textbf{if } m < n \textbf{ then } \chi^{\#1}m \textbf{ else} \\ & \quad \textbf{if } m = n \textbf{ then } \acute{\chi} \\ & \textbf{where } y = \lambda\ m.\ \textbf{if } m < n \textbf{ then } \chi^{\#1}m \textbf{ else} \\ & \quad \textbf{if } m = n \textbf{ then } succ(\chi^{\#2}\chi^{\#4}) \end{aligned}$$

$$\boxed{exit \in G \to G}$$

$$exit\gamma = \lambda\chi.\gamma(\chi^{\#3})$$

$$\boxed{level \in U \to B}$$

$$level\rho = \rho^{\#7}$$

$$\boxed{next \in U \to U}$$

$$next\ \rho = (\rho^{\#1}, \ldots, \rho^{\#5}, \rho^{\#6} + 1, \lambda a.false)$$

7.2. Declarations

$$\boxed{D \in Decl \to U_s \to U \to G \to G}$$

$$\begin{aligned} D[\![const\ \Delta_{c1}; \ldots; \Delta_{cn}]\!]\varsigma\rho\gamma &= \gamma \\ D[\![type\ \Delta_{t1}; \ldots; \Delta_{tn}]\!]\varsigma\rho\gamma &= \gamma \\ D[\![var\ \Delta_{v1}; \ldots; \Delta_{vn}]\!]\varsigma\rho\gamma &= D_v[\![\Delta_{v1}; \ldots; \Delta_{vn}]\!]\varsigma\rho\gamma \\ D[\![procedure\ I(\Pi_1, \ldots, \Pi_n); \Theta]\!]\varsigma\rho\gamma &= \gamma \\ D[\![function\ I(\Pi_1, \ldots, \Pi_n) : T; \Theta]\!]\varsigma\rho\gamma &= \gamma \end{aligned}$$

$$\boxed{D^* \in Decl^* \to U_s \to U \to G \to G}$$

$$D^*[\![\Delta_1; \ldots; \Delta_n]\!]\varsigma\rho\gamma = D^*[\![\Delta_1; \ldots; \Delta_{n-1}]\!]\varsigma\rho; D[\![\Delta_n]\!]\varsigma\rho\gamma$$

$$\boxed{D_v \in V\,dec \to U_s \to U \to G \to G}$$

$$D_v[\![I : T]\!]\varsigma\rho\gamma = \textbf{if } [var : \tau] :: \varsigma[\![I]\!] \textbf{ then } clear\ \tau(\rho[\![I]\!])\gamma$$

$$\boxed{D_v^* \in Vdec^* \to U_s \to U \to G \to G}$$

$$D_v^*[\![\Delta_1;\ldots;\Delta_n]\!]\varsigma\rho\gamma = D_v^*[\![\Delta_{v1};\ldots;\Delta_{vn-1}]\!]\varsigma\rho; D_v[\![\Delta_{vn}]\!]\varsigma\rho\gamma$$

$$\boxed{\mathcal{F} \in Decl \to U_s \to U \to P^*}$$

$$\begin{aligned}
\mathcal{F}[\![const\ \Delta_{c1};\ldots;\Delta_{cn}]\!]\varsigma\rho &= \langle\rangle \\
\mathcal{F}[\![type\ \Delta_{t1};\ldots;\Delta_{tn}]\!]\varsigma\rho &= \langle\rangle \\
\mathcal{F}[\![var\ \Delta_{v1};\ldots;\Delta_{vn}]\!]\varsigma\rho &= \langle\rangle \\
\mathcal{F}[\![procedure\ I(\Pi_1,\ldots,\Pi_n);\Theta]\!]\varsigma\rho &= \langle\pi\rangle \\
&\quad \text{where } \pi = \lambda\gamma.enter(n+1); P^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1\rho_1; \\
&\qquad B[\![\Theta]\!]\varsigma_2\rho_1; exit\ \gamma \\
&\quad \text{where } \varsigma_2 = \varsigma_1[\,[proc{:}\mu_1,\ldots,\mu_n]/I\,] \\
&\text{where } \langle\langle\mu_1,\ldots,\mu_n\rangle,\varsigma_1\rangle = p^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma \\
&\quad \text{where } \rho_1 = \mathcal{Q}^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1(next\ \rho) \\
&\quad \text{where } n = level\ \rho \\
\mathcal{F}[\![function\ I_1(\Pi_1,\ldots,\Pi_n){:}I_2;\Theta]\!]\varsigma\rho &= \langle\pi\rangle \\
&\quad \text{where } \pi = \lambda\gamma.enter(n+1); P^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1\rho_1; \\
&\qquad B[\![\Theta]\!]\varsigma_2\rho_1; return[\![I_1]\!]\rho_2; exit\ \gamma \\
&\quad \text{where } \varsigma_2 = \varsigma_1[\,[afun{:}\mu_1,\ldots,\mu_n{:}\tau]/I_1\,] \\
&\text{where } \langle\langle\mu_1,\ldots,\mu_n\rangle,\varsigma_1\rangle = p^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma \\
&\quad \text{where } \rho_1 = \mathcal{Q}^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma_1(next\ \rho) \\
&\quad \text{where } n = level\ \rho
\end{aligned}$$

$$\boxed{\mathcal{F}^* \in Decl^* \to U_s \to U \to G^*}$$

$$\mathcal{F}^*[\![\Delta_1;\ldots;\Delta_n]\!]\varsigma\rho = \mathcal{F}[\![\Delta_1]\!]\varsigma\rho\cdot\ldots\cdot\mathcal{F}[\![\Delta_n]\!]\varsigma\rho$$

$$\boxed{\mathcal{V} \in Decl \to U_s \to U \to U}$$

$$\begin{aligned}
\mathcal{V}[\![const\ \Delta_{c1};\ldots;\Delta_{cn}]\!]\varsigma\rho &= \rho \\
\mathcal{V}[\![type\ \Delta_{t1};\ldots;\Delta_{tn}]\!]\varsigma\rho &= \rho \\
\mathcal{V}[\![var\ \Delta_{v1};\ldots;\Delta_{vn}]\!]\varsigma\rho &= \mathcal{V}_v^*[\![\Delta_{v1};\ldots;\Delta_{vn}]\!]\varsigma\rho \\
\mathcal{V}[\![procedure\ I(\Pi_1,\ldots,\Pi_n);\Theta]\!]\varsigma\rho &= \rho \\
\mathcal{V}[\![function\ I(\Pi_1,\ldots,\Pi_n){:}T;\Theta]\!]\varsigma\rho &= \rho
\end{aligned}$$

$$\boxed{\mathcal{V}^* \in Decl^* \to U_s \to U \to U}$$

$$\mathcal{V}^*[\![\Delta_1;\ldots;\Delta_n]\!]\varsigma\rho = \mathcal{V}^*[\![\Delta_2;\ldots;\Delta_n]\!]\varsigma;\ \mathcal{V}[\![\Delta_1]\!]\varsigma\rho$$

$$\boxed{\mathcal{V}_v \in Vdec \to U_s \to U \to U}$$

$$\begin{aligned}
\mathcal{V}_v[\![I{:}T]\!]\varsigma\rho &= \text{if } [var{:}\tau]{::}\varsigma[\![I]\!] \text{ then } \dot{\rho}[a/I] \\
&\quad \text{where } (a,\dot{\rho}) = newl\ \tau\rho
\end{aligned}$$

$$\boxed{P \in Par \to U_s \to U \to G \to G}$$

$$\begin{aligned}
P[\![I_1{:}I_2]\!]\varsigma\rho\gamma &= \lambda\epsilon.(update\ \gamma)\epsilon(\rho[\![I_1]\!]) \\
P[\![var\ I_1{:}I_2]\!]\varsigma\rho\gamma &= \lambda\epsilon.(update\ \gamma)\epsilon(\rho[\![I_1]\!])
\end{aligned}$$

$$\boxed{P^* \in Par^* \to U_s \to U \to E^* \to G \to G}$$

$$P^*[\![\Pi_1,\ldots,\Pi_n]\!]\varsigma\rho\gamma = P[\![\Pi_n]\!]\varsigma\rho; P[\![\Pi_{n-1}]\!]\varsigma\rho;\ldots; P[\![\Pi_1]\!]\varsigma\rho\gamma$$

$$\boxed{\mathcal{Q} \in Par \to U_s \to U \to U}$$

$$\begin{aligned}
&\mathcal{Q}[\![var\ I_1{:}I_2]\!]\varsigma\rho = \\
&\quad \mathcal{Q}[\![I_1{:}I_2]\!]\varsigma\rho = \dot{\rho}[a/I_1] \\
&\qquad\qquad \text{where } (a,\dot{\rho}) = new\ \rho
\end{aligned}$$

$$\boxed{\Omega^* \in Par^* \to U_s \to U \to U}$$

$$\Omega^*[\![\Pi_1, \ldots, \Pi_n]\!]\varsigma\rho = \Omega^*[\![\Pi_1, \ldots, \Pi_{n-1}]\!]\varsigma(\Omega[\![\Pi_n]\!]\varsigma\rho)$$

7.3. Expressions

$$\boxed{\mathcal{E}, \hat{\mathcal{E}} \in Exp \to U_s \to U \to G \to G}$$

$$\mathcal{E}[\![E]\!]\varsigma\rho\gamma = \textbf{if } e[\![E]\!]\varsigma :: [const{:}\epsilon{:}\tau] \textbf{ then } apply\ \gamma\epsilon \textbf{ else } \hat{\mathcal{E}}[\![E]\!]\varsigma\rho\gamma$$

$$\hat{\mathcal{E}}[\![N]\!]\varsigma\rho\gamma = apply\ \gamma(value[\![N]\!])$$

$$\hat{\mathcal{E}}[\![I]\!]\varsigma\rho\gamma = \textbf{if } \varsigma[\![I]\!] = [sfun] \textbf{ then } Sf[\![I]\!]\gamma \textbf{ else}$$
$$\textbf{if } \varsigma[\![I]\!] :: [afun{:}\langle\rangle{:}tau] \textbf{ then } (\rho[\![I]\!])^{\#1}\gamma \textbf{ else}$$
$$\textbf{if } \varsigma[\![I]\!] :: [pfun{:}\langle\rangle{:}tau] \textbf{ then } (\rho[\![I]\!])^{\#1}\gamma \textbf{ else } apply\ \gamma(\rho[\![I]\!])$$

$$\hat{\mathcal{E}}[\![O\ E]\!]\varsigma\rho\gamma = \mathcal{R}[\![E]\!]\varsigma\rho; unop[\![O]\!]\gamma$$

$$\hat{\mathcal{E}}[\![E_0\ \Omega\ E_1]\!]\varsigma\rho\gamma = \mathcal{R}[\![E_0]\!]\varsigma\rho; \mathcal{R}[\![E_1]\!]\varsigma\rho; binop[\![\Omega]\!]\gamma$$

$$\hat{\mathcal{E}}[\![E\uparrow]\!]\varsigma\rho\gamma = \mathcal{R}[\![E]\!]\varsigma\rho; verify\ \gamma$$

$$\hat{\mathcal{E}}[\![I(E_1, \ldots, E_n)]\!]\varsigma\rho\gamma = \textbf{if } \varsigma[\![I]\!] = [sfun] \textbf{ then } \mathcal{E}^*[\![E_1, \ldots, E_n]\!]\varsigma\rho; Sf[\![I]\!]\gamma \textbf{ else}$$
$$\textbf{if } \varsigma[\![I]\!] :: [afun{:}\mu_1 \ldots \mu_n{:}\tau] \textbf{ then}$$
$$\mathcal{A}^*[\![E_1, \ldots, E_n]\!]\varsigma\mu_1 \ldots \mu_n\rho; (\rho[\![I]\!])^{\#1}\gamma \textbf{ else}$$
$$\textbf{if } \varsigma[\![I]\!] :: [pfun{:}\mu_1 \ldots \mu_n{:}\tau] \textbf{ then}$$
$$\mathcal{A}^*[\![E_1, \ldots, E_n]\!]\varsigma\mu_1 \ldots \mu_n\rho; (\rho[\![I]\!])^{\#1}\gamma$$

$$\hat{\mathcal{E}}[\![E_0[E_1]]\!]\varsigma\rho\gamma = \mathcal{E}[\![E_0]\!]\varsigma\rho; \mathcal{A}[\![E_1]\!]\varsigma[val{:}\tau_0]\rho; index\ \gamma$$
$$\textbf{where } type(e[\![E_0]\!]\varsigma) :: [\nu{:}array{:}\tau_0{:}\tau_1]$$

$$\hat{\mathcal{E}}[\![E.I]\!]\varsigma\rho\gamma = \mathcal{E}[\![E]\!]\varsigma\rho; select[\![I]\!]\gamma$$

$$\boxed{\mathcal{E}^* \in Exp^* \to U_s \to U \to G \to G}$$

$$\mathcal{E}^*[\![E_1, \ldots, E_n]\!]\varsigma\rho\gamma = \mathcal{E}[\![E_1]\!]\varsigma\rho; \mathcal{E}[\![E_2]\!]\varsigma\rho; \ldots; \mathcal{E}[\![E_n]\!]\varsigma\rho\gamma$$

$$\boxed{\mathcal{L} \in Exp \to U_s \to U \to G \to G}$$

$$\mathcal{L}[\![E]\!]\varsigma\rho\gamma = \textbf{if } e[\![E]\!]\varsigma :: [afun{:}\mu_1 \ldots \mu_n{:}\tau] \textbf{ then } apply\ \gamma(\rho[\![E \mid Id]\!]^{\#2}) \textbf{ else}$$
$$\textbf{if } e[\![E]\!]\varsigma :: [vap{:}\tau] \textbf{ then } \mathcal{E}[\![E]\!]\varsigma\rho; content\ \gamma \textbf{ else}$$
$$\mathcal{E}[\![E]\!]\varsigma\rho\gamma$$

$$\boxed{\mathcal{R} \in Exp \to U_s \to U \to G \to G}$$

$$\mathcal{R}[\![E]\!]\varsigma\rho\gamma = \textbf{if } e[\![E]\!]\varsigma :: [var{:}\tau] \textbf{ then } \mathcal{E}[\![E]\!]\varsigma\rho; content\ \gamma \textbf{ else}$$
$$\textbf{if } e[\![E]\!]\varsigma :: [vap{:}\tau] \textbf{ then } \mathcal{E}[\![E]\!]\varsigma\rho; content; content\ \gamma \textbf{ else}$$
$$\mathcal{E}[\![E]\!]\varsigma\rho\gamma$$

$$\boxed{\mathcal{A} \in Exp \to U_s \to Md \to U \to G \to G}$$

$$\mathcal{A}[\![E]\!]\varsigma\mu\rho\gamma = \textbf{if } \mu :: [var{:}\tau] \textbf{ then } \mathcal{L}[\![E]\!]\varsigma\rho\gamma \textbf{ else}$$
$$\textbf{if } \mu :: [val{:}\tau] \textbf{ then}$$
$$(\ \textbf{if } t :: [\nu{:}sub{:}\epsilon_0{:}\epsilon_1] \textbf{ then}$$
$$\textbf{if } contains\ \tau t \textbf{ then } \mathcal{R}[\![E]\!]\varsigma\rho\gamma$$
$$\textbf{else } \mathcal{R}[\![E]\!]\varsigma\rho; verify_S\ \epsilon_0\epsilon_2\gamma$$
$$\textbf{else } \mathcal{R}[\![E]\!]\varsigma\rho\gamma)$$
$$\textbf{where } t = type(e[\![E]\!]\varsigma)$$

$$\boxed{\mathcal{A}^* \in Exp^* \to U_s \to Md \to U \to G \to G}$$

$$\mathcal{A}^*[\![E_1, \ldots, E_n]\!]\varsigma(\mu_1, \ldots, \mu_n)\rho\gamma = \mathcal{A}[\![E_1]\!]\varsigma\mu_1\rho, \mathcal{A}[\![E_2]\!]\varsigma\mu_2\rho; \ldots; \mathcal{A}[\![E_n]\!]\varsigma\mu_n\rho\gamma$$

7.4. Statements

$$\boxed{C \in Com \to U_s \to U \to G \to G}$$

$$\begin{aligned} C[\![N{:}\Theta]\!]\varsigma\rho\gamma &= B[\![\Theta]\!]\varsigma\rho\gamma \\ C[\![\Theta]\!]\varsigma\rho\gamma &= \mathcal{B}[\![\Theta]\!]\varsigma\rho\gamma \\ C[\![\Gamma_0;\Gamma_1]\!]\varsigma\rho\gamma &= C[\![\Gamma_0]\!]\varsigma\rho; C[\![\Gamma_1]\!]\varsigma\rho\gamma \end{aligned}$$

$$\boxed{C\ell \in Com \to U_s \to U \to C \to C}$$

$$\begin{aligned} C\ell[\![\Gamma]\!]\varsigma\rho\gamma &= C[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma \\ &\quad \text{where } \hat{\varsigma} = \varsigma[(\ldots, true, \ldots)/j[\![\Gamma]\!]], \\ &\quad\ \hat{\rho} = fix\Big(\lambda\dot{\rho}.\rho[J[\![\Gamma]\!]\hat{\varsigma}\dot{\rho}\gamma/j[\![\Gamma]\!]]\Big) \end{aligned}$$

$$\boxed{J \in Com \to U_s \to U \to G \to G^*}$$

$$\begin{aligned} J[\![N{:}\Theta]\!]\varsigma\rho\gamma &= \langle B[\![\Theta]\!]\varsigma\rho\gamma\rangle \\ J[\![\Gamma_1;\Gamma_2]\!]\varsigma\rho\gamma &= J[\![\Gamma_1]\!]\varsigma\rho(C[\![\Gamma_2]\!]\varsigma\rho\gamma)\cdot J[\![\Gamma_2]\!]\varsigma\rho\gamma \\ J[\![\Theta]\!]\varsigma\rho\gamma &= \langle\rangle \end{aligned}$$

$$\boxed{B \in Stm \to U_s \to U \to G \to G}$$

$$B[\![E_0 := E_1]\!]\varsigma\rho\gamma = \mathcal{L}[\![E_0]\!]\varsigma\rho; \mathcal{A}[\![E_1]\!]\varsigma[val{:}\tau]\rho; update\ \gamma$$
$$\text{where } e[\![E_0]\!]\varsigma::[var{:}\tau]$$

$$B[\![if\ E\ then\ \Gamma_0\ else\ \Gamma_1]\!]\varsigma\rho\gamma = \mathcal{R}[\![E]\!]\varsigma\rho; Cond(C\ell[\![\Gamma_0]\!]\varsigma\rho\gamma, C\ell[\![\Gamma_1]\!]\varsigma\rho\gamma)$$

$$B[\![dummy]\!]\varsigma\rho\gamma = \gamma$$

$$B[\![while\ E\ do\ \Gamma\ od]\!]\varsigma\rho\gamma = fix(\lambda\dot{\gamma}.\mathcal{R}[\![E]\!]\varsigma\rho; cond(C\ell[\![\Gamma]\!]\varsigma\rho\dot{\gamma}, \gamma))$$

$$B[\![repeat\ \Gamma\ until\ E]\!]\varsigma\rho\gamma = fix(\lambda\dot{\gamma}.C\ell[\![\Gamma]\!]\varsigma\rho; \mathcal{R}[\![E]\!]\varsigma\rho; cond(\gamma, \dot{\gamma}))$$

$$B[\![goto\ N]\!]\varsigma\rho\gamma = adjust(\rho[\![N]\!])(args\ \gamma - args(\rho[\![N]\!]))$$

$$\begin{aligned} B[\![I(E_1,\ldots,E_n)]\!]\varsigma\rho\gamma = {} & \text{if } \varsigma[\![I]\!]::[sproc] \text{ then } \mathcal{E}^*[\![E_1,\ldots,E_n]\!]\varsigma\rho; Sp[\![I]\!]\gamma \text{ else} \\ & \mathcal{A}^*[\![E_1,\ldots,E_n]\!]\varsigma(\mu_1,\ldots,\mu_n)\rho; \rho[\![I]\!]\gamma \\ & \text{where } (\mu_1,\ldots,\mu_n) = \mathcal{E}^*[\![E_1,\ldots,E_n]\!]\varsigma \end{aligned}$$

$$\begin{aligned} B[\![\Delta^*; begin\ \Gamma\ end]\!]\varsigma\rho\gamma &= D^*[\![\Delta^*]\!]\hat{\varsigma}\hat{\rho}; C[\![\Gamma]\!]\hat{\varsigma}\hat{\rho}\gamma \\ &\quad \text{where } \hat{\varsigma} = d[\![\Delta^*]\!]\varsigma[(\ldots, true, \ldots)/j[\![\Gamma]\!]], \\ &\quad\ \hat{\rho} = fix\Big(\lambda\dot{\rho}.\ddot{\rho}[J[\![\Gamma]\!]\hat{\varsigma}\dot{\rho}\gamma/j[\![\Gamma]\!]][\mathcal{F}^*[\![\Delta^*]\!]\hat{\varsigma}\dot{\rho}/k[\![\Delta^*]\!]]\Big) \\ &\quad \text{where } \ddot{\rho} = \mathcal{V}^*[\![\Delta^*]\!]\hat{\varsigma}\rho \end{aligned}$$

Appendix 2. Formal definition of LT

1. Abstract syntax

$n \in N$	Numerals
$l \in L$	Labels
$p \in P = I_1 \ldots I_n$	Programs
$Cde = N + P + I$	Machine Code
$i \in I =$	
$LIT\ n \mid LOAD \mid ADDR\ n\ n \mid STORE \mid$	
$CALL\ l\ n_1\ n_2 \mid EXIT \mid NEW \mid$	
$NCOND\ l \mid HOP\ l \mid JUMP\ l\ n \mid$	
$LABSET\ l \mid STOP \mid UNOP\ n \mid BINOP\ n \mid$	
$EOF \mid OUTP \mid INPT$	Instructions

2. Semantic Domains

$x \in N$	Integers
$\sigma \in S = V^*$	Stacks
$\rho \in U = L \to G$	Environments
$\delta \in D = N \to D \times N \to N \times D \times N \times G \times N$	Displays
$\mu \in M = (N \to N) \times N^* \times N^* \times N$	Memory
$\gamma \in G = D \to S \to M \to A$	Continuations
$A = N^* + \{eoferror, nostop, \ldots\}$	Answers

3. Auxiliary Functions

$$\boxed{up_T \in D \to G \to N \to N \to N \to D}$$

$$up_T\ \delta\ \gamma\ n\ m\ i = fix(\lambda\dot{\delta}.\langle f, g, \delta, n, \gamma, m\rangle)$$
where $f = \lambda\nu.$ if $\nu < n$ then $\delta^{\#1}\nu$ else if $\nu = n$ then $\dot{\delta}$ else $\perp$
where $g = \lambda\nu.$ if $\nu < n$ then $\delta^{\#2}\nu$ else if $\nu = n$ then k else $\perp$
where $k = \delta^{\#2}\delta^{\#4} + i$

$$\boxed{length \in S \to N}$$

$$length\ \sigma = \text{if } \sigma = x \cdot \dot{\sigma} \text{ then } 1 + length\ \dot{\sigma} \text{ else } 0$$

$$\boxed{down \in S \to N \to S}$$

$$down\ \sigma\ n = \text{if } length\ \sigma = n \text{ then } \sigma \text{ else } down(pop\ \sigma)n$$

$$\boxed{pop \in S \to S}$$

$$pop\ \sigma = \text{if } \sigma = x \cdot \dot{\sigma} \text{ then } \dot{\sigma}$$

4. Semantic Equations

$$\boxed{P \in P \to N^* \to (N^* + E_r)}$$

$$P[\![I_1 \ldots I_n]\!]\ infile = \text{if } distinct(l[\![I_1 \ldots I_n]\!]) \text{ then } \mathcal{M}^*[\![I_1 \ldots I_n]\!]\rho_1\gamma_0\chi_0\sigma_0\mu_0$$
where $\rho_1 = fix(\lambda\rho.\rho_0[\mathcal{L}[\![I_1 \ldots I_n]\!]\rho\gamma_0/l[\![I_1 \ldots I_n]\!]])$

For a program the initial environment, continuation, display, stack, and memory are given as:

$$\rho_0 = \bot$$

$$\gamma_0 = \lambda\delta\sigma\mu.nostop$$

$$\delta_0 = \textbf{fix}\ (\lambda\delta.(\bot[\delta/0], \bot[0/0], \bot, 0, \bot, 0))$$

$$\sigma_0 = \langle\rangle$$

$$\mu_0 = (\bot, infile, \langle\rangle, 0)$$

$$\boxed{\mathcal{L} \in P \to U \to G \to G^*}$$

$$\mathcal{L}[\![I_1, \ldots, I_n]\!]\rho\gamma = \textbf{if}\ I_1{::}[\![LABSET\ l]\!] \textbf{ then } (M^*[\![I_2, \ldots, I_n]\!]\rho\gamma)\cdot\mathcal{L}[\![I_2, \ldots, I_n]\!]\rho\gamma \textbf{ else } \mathcal{L}[\![I_2, \ldots, I_n]\!]\rho\gamma$$

$$\boxed{l \in P \to N^*}$$

$$\ell[\![I_1, \ldots, I_n]\!] = \textbf{if}\ I_1{::}[\![LABSET\ l]\!] \textbf{ then } \langle n\rangle\cdot\ell[\![I_2, \ldots, I_n]\!] \textbf{ else } \ell[\![I_2, \ldots, I_n]\!]$$

$$\boxed{M \in I \to U \to G \to G}$$

$$M[\![LIT\ \ n]\!]\rho\gamma = \lambda\delta\sigma.\gamma\delta(n\cdot\sigma)$$

$$M[\![LOAD]\!]\rho\gamma = \lambda\delta\sigma\mu.\ \textbf{if}\ \sigma{::}x\cdot\dot{\sigma}\ \textbf{then}\ \gamma\delta((\mu x)\cdot\dot{\sigma})\mu$$

$$M[\![ADDR\ x\ \ n]\!]\rho\gamma = \lambda\delta\sigma\mu.\gamma\delta(a\cdot\sigma)\mu \textbf{ where } a = (\delta^{\#2}n) + x$$

$$M[\![STORE]\!]\rho\gamma = \lambda\delta\sigma\mu.\ \textbf{if}\ \sigma{::}x\cdot y\cdot\dot{\sigma}\ \textbf{then}\ \gamma\delta\dot{\sigma}(\mu^{\#1}|x/y|)$$

$$M[\![CALL\ l\ n\ m\ k]\!]\rho\gamma = \lambda\delta\sigma.\rho[\![l]\!](up_T\ \delta\ \gamma\ n(length\ \sigma - m)k)\sigma$$

$$M[\![EXIT]\!]\rho\gamma = \lambda\delta.\dot{\gamma}(\delta^{\#3}) \quad \textbf{where } \dot{\gamma} = \delta^{\#5}$$

$$M[\![NCOND\ l]\!]\rho\gamma = \lambda\delta\sigma.\ \textbf{if}\ \sigma{::}x\cdot\dot{\sigma}\ \textbf{then}\ (\ \textbf{if}\ x \neq 0\ \textbf{then}\ \rho[\![l]\!]\delta\dot{\sigma}\ \textbf{else}\ \gamma\delta\dot{\sigma})$$

$$M[\![HOP\ l]\!]\rho\gamma = \rho[\![l]\!]$$

$$M[\![JUMP\ l\ n]\!]\rho\gamma = \lambda\delta\sigma.\rho[\![x]\!](\delta^{\#1}n)(down\ \sigma(\delta^{\#1}n)^{\#6})$$

$$M[\![LABSET\ l]\!]\rho\gamma = \gamma$$

$$M[\![STOP]\!]\rho\gamma = \lambda\delta\sigma\mu.\mu^{\#3}$$

$$M[\![NEW]\!]\rho\gamma = \lambda\delta\sigma\mu.\gamma\delta(\mu^{\#4}\cdot\sigma)(\mu^{\#1}, \mu^{\#2}, \mu^{\#3}, \mu^{\#4} + 1)$$

$$M[\![UNOP\ \ n]\!]\rho\gamma = \lambda\delta\sigma.\ \textbf{if}\ \sigma{::}x\cdot\dot{\sigma}\ \textbf{then}\ \gamma\delta((O[\![n]\!]x)\cdot\dot{\sigma})$$

$$M[\![BINOP\ \ n]\!]\rho\gamma = \lambda\delta\sigma.\ \textbf{if}\ \sigma{::}x\cdot y\cdot\dot{\sigma}\ \textbf{then}\ \gamma\delta((\mathcal{W}[\![n]\!]y\ x)\cdot\dot{\sigma})$$

$$M[\![EOF]\!]\rho\gamma = \lambda\delta\sigma\mu\ \textbf{if}\ \mu^{\#2} = \langle\rangle\ \textbf{then}\ \gamma\delta(0\cdot\sigma)\mu\ \textbf{else}\ \gamma\delta(1\cdot\sigma)\mu$$

$$M[\![OUTP]\!]\rho\gamma = \lambda\delta\sigma\mu\ \textbf{if}\ \sigma{::}x\cdot\dot{\sigma}\ \textbf{then}\ \gamma\delta\dot{\sigma}\dot{\mu} \quad \textbf{where } \dot{\mu} = (\mu^{\#1}, \mu^{\#2}, \mu^{\#3}\cdot x)$$

$$M[\![INPT]\!]\rho\gamma = \lambda\delta\sigma\mu\ \textbf{if}\ \mu^{\#2}{::}x\cdot\dot{\mu}\ \textbf{then}\ \gamma\delta(x\cdot\sigma)(\mu^{\#1}, \dot{\mu}, \mu^{\#3})\ \textbf{else}\ eoferror$$

O and $\mathcal{W}$ are defined as in LS.

Appendix 3. The Scanner

1. Logical basis

1.1. Definition of the micro syntax

rulefile (*scanner*)

constant *colon, period, equal, null_sequence, idtoken, errortoken, periodtoken, dperiodtoken, numbertoken, length, value, void, syn, sem, tops, names, values, becomestoken, colontoken;*

from *true* infer

$colon \neq period \wedge colon \neq equal \wedge period \neq equal \wedge length \neq value \wedge tops \neq names \wedge syn \neq sem \wedge sem \neq void \wedge syn \neq void$;

%definition of the scanning function %

*psi*1: replace $PSI(tab, s)$ where $eof(s)$ by $null\_sequence$;

*psi*2: replace $PSI(tab, s)$ where $letter(hd(s))$ by $PHIident(tab, list(hd(s)), tl(s))$;

*psi*3: replace $PSI(tab, s)$ where $digit(hd(s))$ by $PHInumber(tab, list(hd(s)), tl(s))$;

*psi*4: replace $PSI(tab, s)$ where $delim(hd(s))$ by $PHIdelim(tab, list(hd(s)), tl(s))$;

*psi*5: replace $PSI(tab, s)$ where $colon = hd(s)$ by $PHIcol(tab, list(hd(s)), tl(s))$;

*psi*6: replace $PSI(tab, s)$ where $period = hd(s)$ by $PHIper(tab, list(hd(s)), tl(s))$;

*psi*7: replace $PSI(tab, s)$ where $\neg letter(hd(s))$

$\wedge \neg digit(hd(s))$
$\wedge \neg delim(hd(s))$
$\wedge \neg colon = hd(s)$
$\wedge \neg period = hd(s)$ by $PSI(tab, tl(s))$;

%scanning identifiers, numbers and delimiters %

*phil*1: replace $PHIident(tab, s1, s2)$ where $letter(hd(s2))$
by $PHIident(tab, concat(s1, list(hd(s2))), tl(s2))$;

*phil*2: replace $PHIident(tab, s1, s2)$ where $digit(hd(s2))$
by $PHIident(tab, concat(s1, list(hd(s2))), tl(s2))$;

*phil*3: replace $PHIident(tab, s1, s2)$ where $\neg digit(hd(s2)) \wedge \neg letter(hd(s2))$ by
$concat(list(Sident(tab, s1)), PSI(STident(tab, s1), s2))$;

*phil*4: replace $PHIident(tab, s1, null\_sequence)$
by $list(Sident(tab, s1))$;

*phic*1: replace $PHIcol(tab, list(colon), null\_sequence)$ by $list(Scol(1))$;

*phic*2: replace $PHIcol(tab, list(colon), s)$ where $hd(s) = equal$
by $concat(list(Scol(2)), PSI(tab, tl(s)))$;

*phic*3: replace $PHIcol(tab, list(colon), s)$ where $hd(s) \neq equal$
by $concat(list(Scol(1)), PSI(tab, s))$;

*phip*1: replace $PHIper(tab, list(period), null\_sequence)$ by $list(Sper(1))$;

*phip*2: replace $PHIper(tab, list(period), s)$ where $hd(s) = period$
by $concat(list(Sper(2)), PSI(tab, tl(s)))$;

*phip*3: replace $PHIper(tab, list(period), s)$ where $hd(s) \neq period$
by $concat(list(Sper(1)), PSI(tab, s))$;

*phid*1: replace $PHIdelim(tab, list(c), s)$ by $concat(list(Sdelim(tab, c)), PSI(tab, s))$;

*phin*1: replace $PHInumber(tab, s1, null\_sequence)$ by $list(Snumber(tab, s1))$;

*phin*2: replace $PHInumber(tab, s1, s2)$ where $digit(hd(s2))$
by $PHInumber(tab, concat(s1, list(hd(s2))), tl(s2))$;

phin3: replace $PHInumber(tab, s1, s2)$ where $\neg digit(hd(s2))$
by $concat(list(Snumber(tab, s1)), PSI(tab, s2))$;

%semantic routines attached to tokens %

Sc1: replace $Scol(1)$ by $mkevoidtoken(colontoken)$;

Sc2: replace $Scol(2)$ by $mkevoidtoken(becomestoken)$;

Sp1: replace $Sper(1)$ by $mkevoidtoken(periodtoken)$;

Sp2: replace $Sper(2)$ by $mkevoidtoken(dperiodtoken)$;

Sn1: replace $Snumber(tab, s)$ by $mketoken(numbertoken, Nscr(s))$;

Sid1: replace $Sident(tab, s)$ where $kappa(s).syn = errortoken$
by $mketoken(idtoken, apply(henter(tab, s), s))$;

Sid2: replace $Sident(tab, s)$ where $kappa(s).syn \neq errortoken$ by $kappa(s)$;

STid1: replace $STident(tab, s)$ where
$kappa(s).syn = errortoken$ by $henter(tab, s)$;

STid4: replace $STident(tab, s)$ where $kappa(s).syn \neq errortoken$ by tab;

Nscr1: replace $Nscr(list(c))$ by $val(c)$;

Nscr2: replace $Nscr(concat(s, list(c)))$ by $10*Nscr(s) + val(c)$;

1.2. Representation functions

hen1: replace $henter(tab, s)$ where $0 = apply(tab, s)$ by $redefine(tab, s)$;

hen2: replace $henter(tab, s)$ where $0 \neq apply(tab, s)$ by tab;

%definition of token representation %

voidt1a: infer $x = mkevoidtoken(a)$ from $x.void \wedge x.syn = a$;

voidt1: whenever $x.void$ from $x.void$ infer $x = mkevoidtoken(x.syn)$;

mtok1a: infer $x = mketoken(a, b)$ from $\neg x.void \wedge x.syn = a \wedge x.sem = b$;

mtok1: whenever $x.void$ from $\neg x.void$ infer $x = mketoken(x.syn, x.sem)$;

%identifier table %

it1: replace $apply(tabrep(t), s2)$ by *cases*
$t.tops = 0 \rightarrow 0$;
$t.tops \neq 0 \wedge stringrep(t.names[t.tops]) = s2 \rightarrow t.tops$;
$t.tops \neq 0 \wedge stringrep(t.names[t.tops]) \neq s2 \rightarrow$
$apply(tabrep(< t, .tops, t.tops - 1 >), s2)$ end ;

it2: replace $newvalue(tabrep(t))$ by $t.tops + 1$;

it3: replace $isitable(tab)$ by $tab.tops \geq 0$;

it4: infer $apply(t1, s) = apply(t2, s)$ from $t1.tops \leq t2.tops \wedge apply(t1, s) \neq 0$;

it5: replace $redefine(tabrep(t), stringrep(s))$ by
$tabrep(<< t, .names[t.tops + 1], s >, .tops, t.tops + 1 >)$;

1.3. Sequences

%various implementation details %

str1: infer $stringrep(z) = null\_sequence$ from $z.length = 0$;

str1a: replace $stringrep(< z, .length, 0 >)$ by $null\_sequence$;

str2: replace $concat(stringrep(z), list(c))$ by
$stringrep(<< z, .length, z.length + 1 >,$
$.value,$
$< z.value, [z.length + 1], c >>)$;

str3: from $stringrep(x) = stringrep(y) \wedge$
$z.value[z.length + 1] = y.value[y.length + 1]$
infer $strinrep(< z, .length, z.length + 1 >) =$
$stringrep(< y, .length, y.length + 1 >)$;

str4: infer $stringrep(x) \neq stringrep(y)$ from $\neg z.length = y.length$;

str5: from $stringrep(< z, .length, i >) \neq stringrep(< y, .length, i >) \wedge$
$i < z.length$

infer $stringrep(x) \neq stringrep(y)$;

str6: infer $stringrep(x) \neq stringrep(y)$ from
$x.value[x.length] \neq y.value[x.length]$;

lst1: replace $hd(list(c))$ by c;

lst2: replace $tl(list(c))$ by $null\_sequence$;

emp1: from *true* infer $eof(null\_sequence)$;

emp2: from $eof(x)$ infer $x = null\_sequence$;

emp3: replace $empty(x)$ by $eof(x)$;

seq1: replace $hd(concat(list(c), s))$ by c;

seq2: replace $tl(concat(list(c), s))$ by s;

seq5: replace $concat(list(hd(x)), tl(x))$ by x;

app1: replace $append(s, c)$ by $concat(s, list(c))$;

seq6: infer $concat(concat(x, y), z) = concat(u, concat(v, w))$ from
$x = u \wedge y = v \wedge z = w$;

seq3: replace $concat(x, y)$ where $eof(x)$ by y;

seq4: replace $concat(x, y)$ where $eof(y)$ by x;

2. The program

pascal

```
const stringlength= 20;
      itabsize= 200;
      errentry= 0;
        period='.';
         colon=':  ';
         equal=' =';
```



```
type tokenkind= (becomestoken, idtoken, numbertoken, periodtoken,
                 errortoken, dperiodtoken, colontoken);
       carray= array [1: stringlength] of charr;
      string= record length: integer; value: carray end ;
       token= record syn: tokenkind; sem: integer; void: boolean end ;
   tokenfile= file of token;
    charfile= file of charr;
      ittype= record tops: integer;
                     names: array [1: itabsize] of string;
                     values: array [1: itabsize] of integer end ;

var cvalid      : boolean;
  infile, infile0: charfile;
       outfile: tokenfile;
             c: charr;
    itab, itab0: ittype;

procedure error;
entry true;
exit false;
  %This procedure will print an error message. Any further action of the
  program will not be covered by the proof; This allows to conclude false
  as an exit condition. %
external ;

function letter(c: charr): boolean;
  %letter(c) iff c ∈ {A, B, ...., x, y, z} no proof for this %
entry true; exit true; external ;

function digit(c: charr): boolean;
  %digit(c) iff c ∈ {0, 1, ...., 9} %
entry true; exit true; external ;

function delim(c: charr): boolean;
  %delim(c) iff c ∈ {;, ∧∨[]() =≠<>≥≤ /...} %
entry true; exit true; external ;

%append a character to a string %
procedure sappend( var s: string; c: charr);
initial s = s0;
entry true;
```

```
exit stringrep(s) = concat(stringrep(s0), list(c));

begin
  if s.length = stringlength then error
  else begin
    s.length ← s.length + 1;
    s.value[s.length] ← c;
  end ;
end ; %sappend %

function stringequal(s1, s2: string): boolean;
entry true; exit stringequal = (stringrep(s1) = stringrep(s2));
var i: integer;
  eq: boolean;

begin
  if s1.length ≠ s2.length then stringequal ← false else
  begin
    i ← 0;
    eq ← true;
    invariant
      ((eq ∧ stringrep(< s1, .length, i >) = stringrep(< s2, .length, i >))
        ∨(¬eq ∧ 1 ≤ i ∧
          stringrep(< s1, .length, i >) ≠ stringrep(< s2, .length, i >)))
      ∧0 ≤ i ∧ i ≤ s1.length ∧ i ≤ s2.length
      ∧s2.length = s1.length
    while (i < (s1.length)) and eq do
    begin
      i ← i + 1;
      if s1.value[i] ≠ s2.value[i] then eq ← false;
    end ;
    stringequal ← eq;
  end ;
end ; %stringequal %

%make a token with void semantic information %
function voidtoken(t: tokenkind): token;
entry true; exit voidtoken = mkevoidtoken(t);
var tok: token;
begin
  tok.syn ← t;
  tok.sem ← 0;
```



```
  tok.void ← true;
  voidtoken ← tok;
end ;

%make a token with semantic information s, nonvoid%
function mtoken(t: tokenkind; s: integer): token;
entry true;
exit mtoken = mketoken(t, s);
var tok: token;
begin
  tok.syn ← t;
  tok.sem ← s;
  tok.void ← false;
  mtoken ← tok;
end ;

procedure senter( var tab: sttype; var r: integer; s: string);
initial  tab = tab0;
entry    isitable(tab);
exit     isitable(tab) ∧
         tabrep(tab) = henter(tabrep(tab0), stringrep(s)) ∧
         r = apply(tabrep(tab), stringrep(s));
var i    : integer;
  found: boolean;

begin
  found ← false;
  i ← 0;
  invariant
    ((¬found ∧ apply(tabrep(< tab, .tops, i >), stringrep(s)) = errentry) ∨
      (found ∧ apply(tabrep(< tab, .tops, i >), stringrep(s)) = i ∧ r = i ∧ r > 0))
    ∧i ≤ tab.tops ∧ 0 ≤ i
  while (i < (tab.tops)) and notfound do
  begin
    i ← i + 1;
    if stringequal(s, tab.names[i]) then
    begin
      r ← i;
      found ← true;
    end ;
  end ;
  if notfound then
```

```
  begin
    tab.tops ← tab.tops + 1;
    tab.names[tab.tops] ← s;
    r ← tab.tops;
  end ;
end ; %ienter %

%scanning identifiers %
procedure scanident;
global ( var cvalid, c, infile, outfile, itab);
initial infile = infile0, outfile = outfile0, c = c0, itab = itab0;
entry  letter(c) ∧ cvalid ∧ isitable(itab);
exit   ((cvalid ∧
       concat(outfile, PSI(tabrep(itab), concat(list(c), infile))) =
       concat(outfile0, PHIident(tabrep(itab0), list(c0), infile0)))
         ∨(¬cvalid ∧ outfile =
       concat(outfile0, PHIident(tabrep(itab0), list(c0), infile0)))) ∧
         (cvalid ∨ eof(infile)) ∧ isitable(itab);
var s  :  string;
    tok:  token;

function keywords(s:  string): token;
entry     true;
exit      keywords = kappa(stringrep(s));
var l, m, r:  integer;
    found:  boolean;
    result:  token;
external ;

procedure mapident( var t:  token; s:  string);
global ( var itab);
initial itab = itab0;
entry  true ∧ isitable(itab);
exit   t = Sident(tabrep(itab0), stringrep(s)) ∧
       tabrep(itab) = STident(tabrep(itab0), stringrep(s)) ∧ isitable(itab);
var i  :  integer;

begin
  t ← keywords(s);
  if errortoken = t.syn then
  begin
    t.syn ← idtoken;
```



```
    ienter(itab, t.sem, s),
    t.void ← false,
  end ;
end ; %mapident %

begin  %body of scanidenit %
  s.length ← 0;
  repeat
    sappend(s, c);
    if eof(infile) then cvalid ← false else read(infile, c)
  until ((notcvalid) or (not(letter(c) or digit(c))))
  invariant {(cvalid ∧
    PHIident(tabrep(itab0), list(c0), infile0) =
        PHIident(tabrep(itab), stringrep(s), concat(list(c), infile)))
    ∨(¬cvalid ∧
    PHIident(tabrep(itab0), list(c0), infile0) =
    list(Sident(tabrep(itab), stringrep(s)))) ∧
    (cvalid ∨ eof(infile));
  mapident(tok, s);
  write(outfile, tok);
end ; %scanident %

%scanning numbers %
procedure scannumber;
global ( var cvalid, c, infile, outfile; itab);
initial infile = infile0, outfile = outfile0, c = c0;
entry  digit(c) ∧ cvalid;
exit   ((cvalid ∧
       concat(outfile, PSI(tabrep(itab), concat(list(c), infile))) =
       concat(outfile0, PHInumber(tabrep(itab), list(c0), infile0)))
         ∨(¬cvalid ∧ outfile =
       concat(outfile0, PHInumber(tabrep(itab), list(c0), infile0)))) ∧
           (cvalid ∨ eof(infile));
var s  :  string;
    v:  integer;

function val(c:  charr): integer;
  %this is ord(c) − ord('0'); take it for granted.. %
entry digit(c); exit true; external ;

begin  %body of scannumber %
  s.length ← 0;
```

```
  v ← 0;
  repeat
    sappend(s, c);
    v ← 10*v + val(c);
    if eof(infile) then cvalid ← false else read(infile, c)
  until notcvalid or notdigit(c)
  invariant ((cvalid ∧
    PHInumber(tabrep(itab), list(c0), infile0) =
        PHInumber(tabrep(itab), stringrep(s), concat(list(c), infile)))
    ∨(¬cvalid ∧
    PHInumber(tabrep(itab), list(c0), infile0) =
    list(Snumber(tabrep(itab), stringrep(s))))) ∧
    (cvalid ∨ eof(infile)) ∧
Nscr(stringrep(s)) = v;
  write(outfile, mtoken(numbertoken, v));
end ; %scannumber %

%scanning delimiters %
procedure scandelim;
global ( var cvalid, c, infile, outfile; itab);
initial infile = infile0, outfile = outfile0, c = c0;
entry  delim(c) ∧ cvalid;
exit   ((cvalid ∧
       concat(outfile, PSI(tabrep(itab), concat(list(c), infile))) =
       concat(outfile0, PHIdelim(tabrep(itab), list(c0), infile0)))
         ∨(¬cvalid ∧ outfile =
       concat(outfile0, PHIdelim(tabrep(itab), list(c0), infile0)))) ∧
           (cvalid ∨ eof(infile));

function mapdelim(tab:  ittype; c:  charr):  token;
entry true;
exit mapdelim = Sdelim(tabrep(tab), c);
external ;

begin
  write(outfile, mapdelim(itab, c));
  if eof(infile) then cvalid ← false else read(infile, c);
end ; %scandelim %

procedure scancol;
global ( var cvalid, c, infile, outfile; itab);
initial infile = infile0, outfile = outfile0, c = c0;
```



```
entry  c = colon ∧ cvalid;
exit   ((cvalid ∧
       concat(outfile, PSI(tabrep(itab), concat(list(c), infile))) =
       concat(outfile0, PHIcol(tabrep(itab), list(c0), infile0)))
         ∨(¬cvalid ∧ outfile =
       concat(outfile0, PHIcol(tabrep(itab), list(c0), infile0)))) ∧
           (cvalid ∨ eof(infile));

function Scol(i:  integer):  token;
entry true; exit true; external ;

begin
  if eof(infile) then
  begin
    cvalid ← false;
    write(outfile, voidtoken(colontoken));
  end  else
  begin
    read(infile, c);
    if c = equal then
    begin
      write(outfile, voidtoken(becomestoken));
      if eof(infile) then cvalid ← false else read(infile, c)
    end  else write(outfile, voidtoken(colontoken));
  end
end ; %scancol %

procedure scanper;
global ( var cvalid, c, infile, outfile; itab);
initial infile = infile0, outfile = outfile0, c = c0;
entry  c = period ∧ cvalid;
exit   ((cvalid ∧
         concat(outfile, PSI(tabrep(itab), concat(list(c), infile))) =
           concat(outfile0, PHIper(tabrep(itab), list(c0), infile0)))
       ∨(¬cvalid ∧
         outfile = concat(outfile0, PHIper(tabrep(itab), list(c0), infile0))))
       ∧(cvalid ∨ eof(infile));

function Sper(i:  integer):  token;
entry true; exit true; external ;

begin
```

```
  if eof(infile) then
  begin
    cvalid ← false;
    write(outfile, voidtoken(periodtoken));
  end else
  begin
    read(infile, c);
    if c = period then
    begin
      write(outfile, voidtoken(dperiodtoken));
      if eof(infile) then cvalid ← false else read(infile, c)
    end else write(outfile, voidtoken(periodtoken));
  end
end ; %scanper %

%main program %
entry eof(outfile) ∧
      infile = infile0 ∧
      ¬eof(infile) ∧
      itab = itab0 ∧
      isitable(itab);
exit  outfile = PSI(tabrep(itab0), infile0);

begin
  read(infile, c);
  cvalid ← true;
  repeat
    if letter(c) then scanident else
    if digit(c) then scannumber else
    if delim(c) then scandelim else
    if c = colon then scancol else
    if c = period then scanper else
    if eof(infile) then cvalid ← false else read(infile, c)
  until not cvalid
  invariant
  (((cvalid ∧ PSI(tabrep(itab0), infile0) =
      concat(outfile, PSI(tabrep(itab), concat(list(c), infile))))
  ∨(¬cvalid ∧ PSI(tabrep(itab0), infile0) = outfile))
  ∧(cvalid ∨ eof(infile))) ∧
  isitable(itab);
end .
```


3. Typical verification conditions

To prove partial correctness of the scanner as presented on the previous page requires to prove 46 verification conditions. Some of these are fairly simple while others require elaborate proofs. All verification conditions can be proven by the Stanford verifier. Here are some typical examples.

Unsimplified Verification Condition: stringequal 4

```
(s1.length ≠ s2.length
→
  false = (stringrep(s1) = stringrep(s2)))
```

Unsimplified Verification Condition: sappend 1

```
(s = s0 ∧
 ¬(s.length = 20) ∧
 s_1 =< s, .length, s.length + 1 > ∧
 s_0 =< s_1,
     .value,
     < s_1.value, [s_1.length], c >>
→
  stringrep(s_0) = concat(stringrep(s0), list(c)))
```

Unsimplified Verification Condition: scanper 2

```
(c = period ∧
 cvalid ∧
 infile = infile0 ∧
 outfile = outfile0 ∧
 c = c0 ∧
 ¬eof(infile)
 →
  ¬empty(infile) ∧
  (infile_2 = rest(infile) ∧
   c_2 = first(infile) ∧
   c_2 = read_x_x(infile, c) ∧
   infile_2 = read_f_f(infile, c) ∧
   c_2 = period ∧
```

$$
\begin{array}{l}
voidtoken(dperiodtoken) = mkevoidtoken(dperiodtoken) \wedge \\
outfile\_2 = append(outfile, voidtoken(dperiodtoken)) \wedge \\
outfile\_2 = write\_f\_f(outfile, voidtoken(dperiodtoken)) \wedge \\
\neg eof(infile\_2) \\
\rightarrow \\
\quad \neg empty(infile\_2) \wedge \\
\quad (infile\_1 = read\_f\_f(infile\_2, c\_2) \wedge \\
\quad c\_1 = read\_z\_z(infile\_2, c\_2) \wedge \\
\quad c\_1 = first(infile\_2) \wedge \\
\quad infile\_1 = rest(infile\_2) \\
\quad \rightarrow \\
\quad\quad (cvalid \wedge \\
\quad\quad concat(outfile\_2, psi(tabrep(itab), concat(list(c\_1), infile\_1))) = \\
\quad\quad\quad concat(outfile0, phiper(tabrep(itab), list(c0), infile0)) \vee \\
\quad\quad \neg cvalid \wedge \\
\quad\quad outfile\_2 = concat(outfile0, phiper(tabrep(itab), list(c0), infile0))) \wedge \\
\quad\quad (cvalid \vee \\
\quad\quad eof(infile\_1)))))
\end{array}
$$

Unsimplified Verification Condition: main 11

$$
\begin{array}{l}
(eof(outfile) \wedge \\
infile = infile0 \wedge \\
\neg eof(infile) \wedge \\
itab = itab0 \wedge \\
isitable(itab) \\
\rightarrow \\
\quad \neg empty(infile) \wedge \\
\quad (infile\_8 = rest(infile) \wedge \\
\quad c\_8 = first(infile) \wedge \\
\quad c\_8 = read\_z\_z(infile, c) \wedge \\
\quad infile\_8 = read\_f\_f(infile, c) \wedge \\
\quad \neg letter(c\_8) \wedge \\
\quad \neg digit(c\_8) \wedge \\
\quad \neg delim(c\_8) \wedge \\
\quad c\_8 = colon \\
\quad \rightarrow \\
\quad\quad true \wedge \\
\quad\quad c\_8 = colon \wedge \\
\quad\quad (outfile\_4 = scancol\_outfile(true, c\_8, infile\_8, outfile, itab) \wedge \\
\quad\quad c\_7 = scancol\_c(true, c\_8, infile\_8, outfile, itab) \wedge
\end{array}
$$

$$
\begin{array}{l}
cvalid\_4 = scancol\_cvalid(true, c\_8, infile\_8, outfile, itab) \wedge \\
infile\_7 = scancol\_infile(true, c\_8, infile\_8, outfile, itab) \wedge \\
(cvalid\_4 \wedge \\
concat(outfile\_4, psi(tabrep(itab), concat(list(c\_7), infile\_7))) = \\
\quad concat(outfile, phicol(tabrep(itab), list(c\_8), infile\_8)) \vee \\
\neg cvalid\_4 \wedge \\
outfile\_4 = concat(outfile, phicol(tabrep(itab), list(c\_8), infile\_8))) \wedge \\
(cvalid\_4 \vee \\
eof(infile\_7)) \\
\rightarrow \\
\quad isitable(itab) \wedge \\
\quad (cvalid\_4 \wedge \\
\quad psi(tabrep(itab0), infile0) = \\
\quad\quad concat(outfile\_4, psi(tabrep(itab), concat(list(c\_7), infile\_7))) \vee \\
\quad \neg cvalid\_4 \wedge \\
\quad psi(tabrep(itab0), infile0) = outfile\_4) \wedge \\
\quad (cvalid\_4 \vee \\
\quad eof(infile\_7)) \wedge \\
\quad (isitable(itab\_0) \wedge \\
\quad\quad (cvalid\_0 \wedge \\
\quad\quad psi(tabrep(itab0), infile0) = \\
\quad\quad\quad concat(outfile\_0, psi(tabrep(itab\_0), concat(list(c\_0), infile\_0))) \vee \\
\quad\quad \neg cvalid\_0 \wedge \\
\quad\quad psi(tabrep(itab0), infile0) = outfile\_0) \wedge \\
\quad\quad (cvalid\_0 \vee \\
\quad\quad eof(infile\_0)) \wedge \\
\quad\quad \neg\neg\neg cvalid\_0 \\
\quad\quad \rightarrow \\
\quad\quad\quad outfile\_0 = psi(tabrep(itab0), infile0)))))
\end{array}
$$

Appendix 4. The Parser

1. Logical basis

In this section we give the theory necessary to verify the parser. Several details (e.g. about sequences) are omitted.

1.1. Representation functions

1.2. LR theory

All parsing actions are distinct

$$\begin{aligned}
&accept \neq reduce \wedge \\
&accept \neq shiftreduce \wedge \\
&accept \neq error \wedge \\
&accept \neq shift \wedge \\
&reduce \neq shiftreduce \wedge \\
&reduce \neq error \wedge \\
&reduce \neq shift \wedge \\
&shiftreduce \neq error \wedge \\
&shiftreduce \neq shift \wedge \\
&shift \neq error;
\end{aligned}$$

$$x \neq error \wedge x \neq shift \wedge x \neq reduce \wedge x \neq shiftreduce \Rightarrow x = accept$$

Definition of *isderiv*

$$\begin{aligned}
&isderiv(mkeforest(x), x) \\
&isderiv(u{\cdot}ts{\cdot}v, w) \wedge roots(ts) = rhs(production(l)) \\
&\quad \Rightarrow isderiv(u{\cdot}(l){\cdot}v, w) \\
&isderiv(v, w) \wedge \langle\rangle = rhs(production(l)) \\
&\quad \Rightarrow isderiv((l){\cdot}v, w)
\end{aligned}$$

Where *production*(l) means $P\ l$ and *rhs* selects the right hand side of a production (l, r).

Definition of slrrel

$$\begin{aligned}
&slrrel(\langle initial_state\rangle, \langle\rangle) \\
&slrrel(u, v) \wedge s = slr(hd(last(1, u)), n).state \\
&\quad \Rightarrow slrrel(u{\cdot}\langle s\rangle, v{\cdot}\langle n\rangle)
\end{aligned}$$

A derived lemma

$$slrrel(remain(n, s), remain(n, nt))$$

Properties of Lr-parsing tables

$$\begin{aligned}
&slrrel(st, nt) \wedge slr(s, x).skind = reduce \wedge list(s) = last(1, st) \\
&\quad \rightarrow last(length(slr(s, x).prod), nt) = rhs(slr(s, x).prod) \\
&slrrel(st, nt) \wedge slr(s, x).skind = shiftreduce \wedge s = hd(last(1, st)) \\
&\quad \rightarrow last(length(slr(s{\cdot}x).prod) - 1, nt), list(x) = rhs(slr(s, x).prod) \\
&slrrel(st, nt) \wedge slr(hd(last(1, st)), x.syn).skind = accept \\
&\quad \rightarrow nt{\cdot}list(x.syn) = rhs(slr(hd(last(1, st)), x.syn).prod) \wedge \\
&\qquad len(nt) = 1 \wedge \\
&\qquad length(slr(hd(last(1, st)), x.syn).prod) = 2 \wedge \\
&\qquad lhs(slr(hd(last(1, st)), x.syn).prod) = startsymbol \wedge \\
&\qquad x = eof_symbol \\
&true \rightarrow slr(st, lhs(p)).skind \neq accept
\end{aligned}$$

1.3. Tree transformations

element(x, n) selects element number n from sequence x. It is defined in terms of *astseqrep*.

$$element(astseqrep(rc, ar, s, l), n) = synrep(rc, ar[s + n - 1])$$

Some of the definitional clauses for *trtr* are given below. The remaining clauses are obvious from the definition of Ξ.

$$trtr(z\_1, rhs) = element(rhs, 1)$$
$$trtr(prog\_1, rhs) = mkeprogram(element(rhs, 2))$$
$$trtr(cdefp\_1, rhs) = mkenulllist$$
$$trtr(cdefp\_2, rhs) = mkeconst(element(rhs, 2))$$
$$trtr(cdefl\_1, rhs) = mkelist(element(rhs, 1))$$
$$trtr(cdefl\_2, rhs) = mkeappend(element(rhs, 1), mkelist(element(rhs, 2)))$$
$$trtr(cdef\_1, rhs) = mkepair(element(rhs, 1), element(rhs, 3))$$
$$trtr(tdefp\_1, rhs) = mkenulllist$$
$$trtr(tdefp\_2, rhs) = mketype(element(rhs, 2))$$
$$trtr(tdefl\_1, rhs) = mkelist(element(rhs, 1))$$
$$trtr(tdefl\_2, rhs) = mkeappend(element(rhs, 1), mkelist(element(rhs, 2)))$$
$$trtr(tdef\_1, rhs) = mkepair(element(rhs, 1), element(rhs, 3))$$
$$trtr(tden\_1, rhs) = mkeenum(element(rhs, 2))$$

1.4. Extension operations

Definition: subclass

$$subclass(x, x);$$
$$subclass(x, x \cup p);$$
$$subclass(x, z) \wedge subclass(z, y) => subclass(x, y)$$

A derived lemma assuming standard interpretation of *pointer_to*.

$$subclass(rc1, rc2) \wedge \neg pointer\_to(p, rc1) => subclass(rc1, < rc2, \subset p \supset, e >)$$

Definition of *proper*

$$pointer\_to(p, rc) => proper(synrep(rc, p))$$
$$proper(astseqrep(rc, ar, s, f)) \wedge s \leq i \wedge i \leq s + f - 1 => proper(synrep(rc, ar[i]))$$
$$subclass(r, q) \wedge proper(synrep(r, p)) => proper(synrep(q, p))$$
$$empty(z) => proper(z)$$
$$subclass(r, q) \wedge proper(astseqrep(r, p, f, t)) => proper(astseqrep(q, p, f, t))$$
$$proper(synrep(ar, pt)) \wedge proper(astseqrep(rc, ar, s, f)) => proper(astseqrep(rc, < ar, [i], pt >, s, f))$$
$$proper(astseqrep(rc, ar, s, f - 1)) \wedge proper(synrep(rc, ar[s + f - 1])) => proper(astseqrep(rc, ar, s, f))$$
$$proper(astseqrep(rc, a, m, n)) \wedge subclass(rc, pc) => astseqrep(rc, a, m, n) = astseqrep(pc, a, m, n)$$
$$proper(astseqrep(rc, ar, sprime, fprime)) \wedge sprime \leq s \wedge f \leq fprime => proper(astseqrep(rc, ar, s, f))$$

2. The program

```
pascal

const stacklen = 200;

type
  abstract_functions= (mkedummy, mkenulllist);
            plabel= (termprod,
                  z_1, prog_1, block_1, cdefp_1,
                  cdefp_2, cdefl_1, cdefl_2, cdef_1,
                  tdefp_1, tdefp_2, tdefl_1, tdefl_2,
                  tdef_1, tden_1, tden_2, tden_3,
                  tden_4, tden_5, tden_6, tden_7,
                  idl_1, idl_2, vdecl_1. vdecl_2,
                  vdec_1, vdecp_1, vdecp_2, pfdecp_1,
                  pfdecp_2, pfdecp_3, fdec_1,
                  pdec_1, cstmt_1, com_1, com_2, com_3,
                  stmt_1, stmt_2, stmt_3, stmt_4,
                  stmt_5, stmt_6, stmt_7, stmt_8, stmt_9,
                  stmt_10, expr_1, expr_2,
                  conj_1, conj_2, conj_3, rel_1, rel_2,
                  rel_3, sum_1, sum_2, sum_3,
                  term_1, term_2, fact_1, fact_2, fact_3,
                  fact_4, fact_5, fact_6,
                  vble_1, vble_2, vble_3, vble_4, parms_1,
                  parms_2, parml_1, parml_2,
                  parm_1, parm_2, exprl_1, exprl_2,
                  idn_1, idn_2);

              tree= 1:2;
      termnonterm= (startsymbol);
           token= record
                    syn:termnonterm;
                    sem:integer;
                    void:boolean end ;
       tokenfile= file of token;
       asynkind= (tagterminal,
                  tagappend, tagarray, tagassign,
                  tagbinop, tagblock, tagcommandlist,
                  tagconst, tagdcref, tagdummy,
                  tagenum, tagfcall, tagfunction,
```

```
                    taggoto, tagif, tagindex,
                    taglabel, tagtypedec, tagconstdec,
                    tagvardec, tagpcall, tagpointer,
                    tagprocedure, tagprogram, tagrecord,
                    tagrepeat, tagselect, tagstmt,
                    tagsubrange, tagtype, tagtypeid,
                    tagunop, tagvalp, tagvard,
                    tagvarp, tagwhile);

          atree=↑ anode;
         anode=  record skind:asynkind;
                    subt1:atree;
                    subt2:atree;
                    subt3:atree;
                    subt4:atree;
                    next:atree;
                    info:integer end ;
       stateset= 1:2;
    astsequence=  array [1:stacklen]of atree;
   treesequence= (null_sequence);
     ntsequence= (b1, b2);
  statesequence=  array [1:stacklen]of stateset;
         action=  record
                    skind:(accept, error, shift, reduce, shiftreduce);
                    prod:plabel;
                    state:stateset end ;

var
    tlst, treestack: treesequence;
    slst, statestackptr: integer;
    nlst, ntstack: ntsequence;
    alst, aststackptr: integer;
    aststack: astsequence;
    statestack: statesequence;
    input: tokenfile;
    input0: ntsequence;
    act: action;
    l, m, n: integer;
    nt: termnonterm;
    initial_state: stateset;
    eof_symbol: token;
    parsetree: tree;
```



```
    syntree: tree;
    ptr, ast: atree;
    look: token;

%Building abstract syntax trees %

procedure Cmkefunction(p1, p2, p3, p4: atree; var result: atree);
global ( var #anode);
initial #anode = anode0;
entry  proper(synrep(#anode, p1)) ∧ proper(synrep(#anode, p2)) ∧
       proper(synrep(#anode, p3)) ∧ proper(synrep(#anode, p4));
exit   mkefunction(synrep(#anode, p1), synrep(#anode, p2),
       synrep(#anode, p3), synrep(#anode, p4)) = synrep(#anode, result) ∧
       subclass(anode0, #anode) ∧
       proper(synrep(#anode, result));
begin
 new(result);
 result ↑ .skind ← tagfunction;
 result ↑ .subt1 ← p1;
  comment proper(synrep(#anode, result)) ∧
          proper(synrep(#anode, p1)) ∧ proper(synrep(#anode, p2)) ∧
          proper(synrep(#anode, p3)) ∧ proper(synrep(#anode, p4)) ∧
          result ↑ .skind = tagfunction ∧ result ↑ .subt1 = p1 ∧
          subclass(anode0, #anode);
 result ↑ .subt2 ← p2;
 result ↑ .subt3 ← p3;
  comment proper(synrep(#anode, result)) ∧
          result ↑ .skind = tagfunction ∧ result ↑ .subt1 = p1 ∧
          result ↑ .subt2 = p2 ∧
          result ↑ .subt3 = p3 ∧ proper(synrep(#anode, p4)) ∧
          subclass(anode0, #anode);
 result ↑ .subt4 ← p4;
end ;

                              ...

          < one function for each clause of abstract syntax >

                              ...

procedure Cmkeappend(p1, p2: atree; var result: atree);
global ( var #anode);
```

```
initial #anode = anode0;
entry  proper(synrep(#anode, p1)) ∧ proper(synrep(#anode, p2)) ∧
       islist(synrep(#anode, p1)) ∧ islist(synrep(#anode, p2));
exit   mkeappend(synrep(#anode, p1), synrep(#anode, p2))
         = synrep(#anode, result) ∧
       subclass(anode0, #anode) ∧
       islist(synrep(#anode, result)) ∧
       proper(synrep(#anode, result));
var
    t1: atree;
begin
  if p1 = nil then result ← p2 else
  begin
    new(result);
    result ↑ .subt1 ← p1 ↑ .subt1;
     comment subclass(anode0, #anode);
    Cmkeappend(p1 ↑ .nezt, p2, t1);
    result ↑ .nezt ← t1;
     comment synrep(#anode, #anode ⊂ p1 ⊃ .nezt) =
                  selrest(synrep(#anode, p1)) ∧
             isnulllist(synrep(#anode, p1)) = truthrep(false) ∧
             synrep(#anode, #anode ⊂ p1 ⊃ .subt1) =
                  selfirst(synrep(#anode, p1));
  end ;
end ;

                        ...

                        ...

procedure Cmkefcall(p1, p2: atree; var result: atree);
global ( var #anode);
initial #anode = anode0;
entry  proper(synrep(#anode, p1)) ∧ proper(synrep(#anode, p2));
exit   mkefcall(synrep(#anode, p1), synrep(#anode, p2)) =
           synrep(#anode, result) ∧
       subclass(anode0, #anode) ∧
       proper(synrep(#anode, result));
begin
  new(result);
  result ↑ .skind ← tagfcall;
  result ↑ .subt1 ← p1;
```



```
  result ↑ .subt2 ← p2;
end ;

%ctrtr implements trtr %

procedure ctrtr(p: plabel; tstart, tlen: integer; var result: atree);
global ( var #anode; aststack);
initial #anode = #anode0;
entry  len(astseqrep(#anode, aststack, tstart, tlen)) = length(p) ∧
       proper(astseqrep(#anode, aststack, tstart, tlen));
exit   synrep(#anode, result) =
           trtr(p, astseqrep(#anode, aststack, tstart, tlen)) ∧
       proper(synrep(#anode, result)) ∧
       subclass(#anode0, #anode);

var t1, t2: atree;
begin

  case p of

     z_1:   %s ::= prog eofsymbol %
          result ← aststack[tstart];

          ...

          < one case for each label of G >

          ...

  stmt_5:   %stmt ::= ifsymbol expr thensymbol com fisymbol %
          begin
            Cmkedummy(t1);
            Cmkestmt(t1, t2);
            Cmkeif(aststack[tstart + 1], aststack[tstart + 3], t2, result);
          end;

          ...

    idn_2:   %idn ::= numbersymbol %
          result ← aststack[tstart] end ;

end ;
```

```
%termtr does tree transformations for terminals %

procedure Ctermtr(t: token; var result: atree);
global ( var #anode);
initial #anode = #anode0;
entry  true;
exit   synrep(#anode, result) = termtr(t) ∧
       proper(synrep(#anode, result)) ∧
       subclass(#anode0, #anode);
begin
  new(result);
  result ↑ .skind ← tagterminal;
  result ↑ .info ← t.sem;
end ;


procedure tnpop( var t: treesequence; n: integer; var tlst: treesequence);
initial t = t0;
entry  true;
exit   t = remain(n, t0) ∧ tlst = last(n, t0) ∧ t0 = concat(t, tlst);
external ;

procedure nnpop( var t: ntsequence; n: integer; var tlst: ntsequence);
initial t = t0;
entry  true;
exit   t = remain(n, t0) ∧ tlst = last(n, t0) ∧ t0 = concat(t, tlst);
external ;

procedure tpush( var ts: treesequence; t: tree);
initial ts = ts0;
entry  true;
exit   ts = concat(ts0, list(t));
external ;

procedure npush( var ts: ntsequence; t: termnonterm);
initial ts = ts0;
entry  true;
exit   ts = concat(ts0, list(t));
external ;

procedure tclear( var t: treesequence);
```



```
entry true; exit empty(t); external ;

procedure nclear( var t: ntsequence);
entry true; exit empty(t); external ;

function slr(s: stateset; x: termnonterm): action;
entry true; exit true; external ;

procedure error;
entry true; exit false; external ;

function length(p: plabel): integer;
entry true; exit length ≥ 0; external ;

function lhs(p: plabel): termnonterm;
entry true; exit true; external ;

function append(s: treesequence; t: tree): treesequence;
entry true; exit append = concat(s, list(t)); external ;


function mketree(p: plabel;
                 nl: termnonterm;
                 tl: treesequence;
                 sm: integer): tree;
entry true; exit true; external ;

procedure apush( var s: integer; x: atree);
global ( var aststack; #anode);
initial s = s0, aststack = aststack0, #anode = #anode0;
entry  proper(astseqrep(#anode, aststack, 1, s)) ∧ proper(synrep(#anode, x));
exit   astseqrep(#anode, aststack, 1, s) =
       concat(astseqrep(#anode, aststack0, 1, s0), list(synrep(#anode, x))) ∧
       proper(astseqrep(#anode, aststack, 1, s));
begin
  s ← s + 1;
  aststack[s] ← x;
end ;

%main parsing loop %

entry concat(input, list(eof_symbol)) = input0 ∧
```

```
      empty(stateseqrep(statestack, 1, statestackptr)) ∧
      uniqueeof(input) ∧
      empty(astseqrep(#anode, aststack, 1, aststackptr));
exit  isderiv(list(parsetree), input0) ∧ root(parsetree) = startsymbol ∧
      treetr(parsetree) = synrep(#anode, ast);
begin
  tclear(treestack);
  nclear(ntstack);
  spush(statestackptr, initial_state);
  if eof(input) then look ← eof_symbol else read(input, look);

  assert slrrel(stateseqrep(statestack, 1, statestackptr), ntstack) ∧
         isderiv(mkeforest(seqrep(look, input)), input0) ∧
         treetrs(treestack) = astseqrep(#anode, aststack, 1, aststackptr) ∧
         roots(treestack) = ntstack ∧
         empty(ntstack) ∧
         stateseqrep(statestack, 1, statestackptr) = list(initial_state) ∧
         input0 = seqrep(look, input) ∧
         uniqueeof(input) ∧ empty(treestack) ∧
         proper(astseqrep(#anode, aststack, 1, aststackptr)) ∧
         len(treestack) = len(ntstack);
  repeat
    act ← slr(stop(statestackptr), look.syn);
    Ctermtr(look, ptr);
    syntree ← mketree(termprod, look.syn, null_sequence, look.sem);
    nt ← look.syn;

    if act.skind = error then error else
    if act.skind = shift then
    begin
      spush(statestackptr, act.state);
      npush(ntstack, look.syn);
      apush(aststackptr, ptr);
      tpush(treestack, syntree);
      get(look);
    end
    else
    if act.skind ≠ accept then begin
      if act.skind = reduce then
      begin
        n ← length(act.prod);
        nt ← lhs(act.prod);
```



```
        tnpop(treestack, n, tlst);
        anpop(aststackptr, n, alst);
        nnpop(ntstack, n, nlst);
        snpop(statestackptr, n);
        comment nlst = rhs(act.prod) ∧
                slrrel(stateseqrep(statestack, 1, statestackptr), ntstack) ∧
                roots(tlst) = nlst ∧
                len(astseqrep(#anode, aststack, alst, n)) = length(act.prod) ∧
                treetrs(tlst) = astseqrep(#anode, aststack, alst, n);
        ctrtr(act.prod, alst, n, ptr);
        syntree ← mketree(act.prod, nt, tlst, 0);
        act ← slr(stop(statestackptr), nt);
      end else begin
        nt ← look.syn;
        get(look);
      end ;
      invariant
              len(treestack) = len(ntstack) ∧
              roots(treestack) = ntstack ∧
              root(syntree) = nt ∧
              treetrs(treestack) =
              astseqrep(#anode, aststack, 1, aststackptr) ∧
              proper(astseqrep(#anode, aststack, 1, aststackptr)) ∧
              isderiv(concat(concat(treestack, list(syntree)),
                mkeforest(seqrep(look, input))),
                  input0) ∧
              synrep(#anode, ptr) = treetr(syntree) ∧
              proper(synrep(#anode, ptr)) ∧
              slrrel(stateseqrep(statestack, 1, statestackptr), ntstack) ∧
              act.skind ≠ accept ∧
              act = slr(hd(last(1,
                stateseqrep(statestack, 1, statestackptr))),
                  root(syntree))
      while act.skind = shiftreduce do
      begin
        n ← length(act.prod);
        tnpop(treestack, n − 1, tlst);
        anpop(aststackptr, n − 1, alst);
        nnpop(ntstack, n − 1, nlst);
        snpop(statestackptr, n − 1);
        comment
                concat(nlst, list(nt)) = rhs(act.prod) ∧
```

```
                slrrel(stateseqrep(statestack, 1, statestackptr), ntstack) ∧
                roots(tlst) = nlst ∧
                treetrs(tlst) = astseqrep(#anode, aststack, alst, n − 1) ∧
                len(astseqrep(#anode, aststack, alst, n − 1)) =
                    length(act.prod) − 1 ∧
                len(astseqrep(#anode, aststack, alst, n − 1)) = len(tlst) ∧
                len(tlst) = len(nlst) ∧
                isderiv(concat(concat(concat(treestack, tlst),
                  list(syntree)),
                    mkeforest(seqrep(look, input))),
                      input0) ∧
                treetrs(concat(tlst, list(syntree))) =
                  concat(astseqrep(#anode, aststack, alst, n − 1),
                    list(synrep(#anode, ptr)));
      aappend(alst, n − 1, ptr, l);
      ctrtr(act.prod, alst, l, ptr);
      nt ← lhs(act.prod);
      syntree ← mketree(act.prod, nt, append(tlst, syntree), 0);
       comment
                treetr(syntree) = synrep(#anode, ptr);
                act ← slr(stop(statestackptr), nt);
    end ;
    tpush(treestack, syntree);
    apush(aststackptr, ptr);
    npush(ntstack, nt);
    spush(statestackptr, act.state);
  end ;
until act.skind = accept
invariant
        uniqueeof(input) ∧
        isderiv(concat(treestack, mkeforest(seqrep(look, input))), input0) ∧
        slrrel(stateseqrep(statestack, 1, statestackptr), ntstack) ∧
        treetrs(treestack) = astseqrep(#anode, aststack, 1, aststackptr) ∧
        proper(astseqrep(#anode, aststack, 1, aststackptr)) ∧
        roots(treestack) = ntstack ∧
        len(treestack) = len(ntstack) ∧
        (act.skind = accept →
        (act = slr(hd(last(1, stateseqrep(statestack, 1, statestackptr))), nt) ∧
        nt = look.syn));
begin
  n ← length(act.prod);
  nt ← lhs(act.prod);
```



```
    tnpop(treestack, n − 1, tlst);
    anpop(aststackptr, n − 1, alst);
    nnpop(ntstack, n − 1, nlst);
    snpop(statestackptr, n − 1);
     assert concat(nlst, list(look.syn)) = rhs(act.prod) ∧
          seqrep(look, input) = list(eof_symbol) ∧
          slrrel(stateseqrep(statestack, 1, statestackptr), ntstack) ∧
          len(treestack) = len(ntstack) ∧
          roots(tlst) = nlst ∧ len(treestack) = 0 ∧
          len(astseqrep(#anode, aststack, alst, n − 1)) = length(act.prod) − 1 ∧
          len(astseqrep(#anode, aststack, 1, aststackptr)) = 0 ∧
          treetrs(tlst) = astseqrep(#anode, aststack, alst, n − 1) ∧
          nt = startsymbol ∧
          proper(astseqrep(#anode, aststack, alst, n − 1)) ∧
          isderiv(concat(tlst,
            mkeforest(list(look))),
              input0);
    Ctermtr(look, ptr);
    aappend(alst, n − 1, ptr, l);
    ctrtr(act.prod, alst, l, ast);
    parsetree ← mketree(act.prod, nt,
      append(tlst,
        mketree(termprod, look.syn, null_sequence, look.sem)), 0);
  end ;
end .
```

Appendix 5. Static semantics

1. Logical basis

In this section we give some of the rules necessary for the verification of the static semantics. We first present the formal definition of *e* in the verifier's rule language. The translation of other definitions into machine readable form follows analogously.

In addition to the definiton of all semantics functions operationalization of the fixed point defining recursive types is required for the proof of the static semantics. This theory is also presented below.

1.1. Rules for *e*

rulefile (*static_e*)

constant $TT, FF, UU, int, bool$;

*ssem*32: replace $Ae(mkenumber(n), zeta)$ by
$mkeconstmode(An(n), mkesubtype(tagrep(int), An(n), An(n)))$;

*ssem*33: replace $Ae(mkeezpid(I), zeta)$ where
$FF = istypemode(apply1(zeta, I)) \wedge$
$FF = isprocmode(apply1(zeta, I)) \wedge$
$FF = issprocmode(apply1(zeta, I))$ by $apply1(zeta, I)$;

*ssem*37: replace $Ae(mkeunop(O, E), zeta)$ by $Ao(O, Ae(E, zeta))$;

*ssem*38: replace $Ae(mkebinop(E0, Omega, E1), zeta)$ by
$Aw(Omega, Ae(E0, zeta), Ae(E1, zeta))$;

*ssem*39: replace $Ae(mkefcall(I, Elist), zeta)$ where
$TT = isafuncmode(apply1(zeta, I)) \wedge$

$apply1(zeta, i) = mkeafuncmode(mulist, tau) \wedge$
$TT = passablestar(mulist, Aestar(Elist, zeta))$ by $mkevalmode(tau)$;

*ssem*40: replace $Ae(mkefcall(I, Elist), zeta)$ where
$TT = ispfuncmode(apply1(zeta, I)) \wedge$
$apply1(zeta, i) = mkepfuncmode(mulist, tau) \wedge$
$TT = passablestar(mulist, Aestar(Elist, zeta))$ by $mkevalmode(tau)$;

*ssem*41: replace $Ae(mkefcall(I, Elist), zeta)$ where
$TT = issfuncmode(apply1(zeta, I))$ by $Asf(I, Aestar(Elist, zeta))$;

*ssem*42: replace $Ae(mkeselect(E, I), zeta)$ where
$Ae(E, zeta) = mkevarmode(mkerecordtype(nu, Psi)) \wedge$
$TT = iselement(mkepair(I, mu), Psi)$ by mu;

*ssem*43: replace $Ae(mkeselect(E, I), zeta)$ where
$FF = isvarmode(Ae(E, zeta)) \wedge$
$Ae(E, zeta) = mkevapmode(mkerecordtype(nu, Psi)) \wedge$
$TT = iselement(mkepair(I, mu), Psi)$ by mu;

*ssem*44: replace $Ae(mkeindex(E0, E1), zeta)$ where
$Atype(Ae(E0, zeta)) = mkearraytype(nu, tau1, tau2) \wedge$
$TT = compatible(tau1, (Atype(Ae(E1, zeta)))) \wedge$
$TT = isvar(Ae(E0, zeta))$ by $mkevarmode(tau2)$;

*ssem*45: replace $Ae(mkeindex(E0, E1), zeta)$ where
$Atype(Ae(E0, zeta)) = mkearraytype(nu, tau1, tau2) \wedge$
$TT = compatible(tau1, (Atype(Ae(E1, zeta)))) \wedge$
$FF = isvar(Ae(E0, zeta)) \wedge$
$TT = isval(Ae(E0, zeta))$ by $mkevalmode(tau2)$;

*ssem*46: replace $Ae(mkederef(E), zeta)$ where
$Atype(Ae(E, zeta)) = mkepointertype(nu, tau)$
by $mkevarmode(tau)$;

*ssem*47: replace $Aestar(Elist, zeta)$ by
$mkeappend(mkelist(Ae(selfirst(Elist), zeta)),$
$Aestar(selrest(Elist), zeta))$;

*ssem*48: replace $Aestar(x, zeta)$
where $isnulllist(x) = TT$ by $mkenulllist$;

1.2. Recursive types

The function symbol At denotes the semantics function t; Ft denotes the function t_f used in the operationalization. We first present the definition of these two functions in the rule language and then present lemmas dealing with least fixed points.

1.2.1. Types

rulefile ($static\_t$)

%static semantics for types %

constant TT, FF, UU;

$st1$: replace $At(mketypeid(I), zeta0, zeta1)$ where
$apply1(zeta0, I) = mketypemode(tau)$ by
$mkepair(tau, zeta0)$;

$Fst1$: replace $Ft(mketypeid(I), zeta0, ur)$ where
$apply1(zeta0, I) = mketypemode(tau)$ by
$mketriple(tau, zeta0, ur)$;

%Enumeration %

$st2$: replace $At(mkeenum(ilst), zeta0, zeta1)$ where
$mkepair(zeta3, n) = etstar(ilst, nu, zeta2, intrep(0)) \wedge$
$newtag(zeta0) = mkepair(nu, zeta2)$ by
$mkepair(mkesubtype(nu, intrep(1), n), zeta3)$;

$Fst2$: replace $Ft(mkeenum(ilst), zeta0, ur)$ where
$mkepair(zeta3, n) = etstar(ilst, nu, zeta2, intrep(0)) \wedge$
$newtag(zeta0) = mkepair(nu, zeta2)$ by
$mketriple(mkesubtype(nu, intrep(1), n), zeta3, ur)$;

%where we introduced and auxiliary function $etstar$ as defined below: %

$et1$: replace $etstar(ilst, nu, zeta, n)$ where $isnulllist(ilst) = TT$
by $mkepair(zeta, n)$;

$et2$: replace $etstar(ilst, nu, zeta, n)$ where $isnulllist(ilst) = FF$
by $etstar(selrest(ilst), nu, et(selfirst(ilst), nu, zeta, succ(n)), succ(n))$;

$et3$: replace $et(id, nu, zeta, n)$ by
$redefine(zeta, mkeconstmode(n, mkesubtype(nu, n, n)), id)$;

$succ1$: replace $succ(intrep(n))$ by $intrep(n + 1)$;

$st3$: replace $At(mkesubrange(E1, E2), zeta0, zeta1)$ where
$isconstmode(Ae(E1, zeta0)) = TT \wedge$
$isconstmode(Ae(E2, zeta0)) = TT$ by
$mkepair(Aunion(Atype(Ae(E1, zeta0)), Atype(Ae(E2, zeta0))), zeta0)$;

$Fst3$: replace $Ft(mkesubrange(E1, E2), zeta0, ur)$ where
$isconstmode(Ae(E1, zeta0)) = TT \wedge$
$isconstmode(Ae(E2, zeta0)) = TT$ by
$mketriple(Aunion(Atype(Ae(E1, zeta0)), Atype(Ae(E2, zeta0))), zeta0, ur)$;

$st4$: replace $At(mkearray(T1, T2), zeta0, zeta1)$ where
$At(T1, zeta0, zeta1) = mkepair(tau1, zeta2) \wedge$
$At(T2, zeta2, zeta1) = mkepair(tau2, zeta3) \wedge$
$issubtype(tau1) = TT \wedge$
$newtag(zeta3) = mkepair(nu, zeta4)$ by
$mkepair(mkearraytype(nu, tau1, tau2), zeta4)$;

$Fst4$: replace $Ft(mkearray(T1, T2), zeta0, ur)$ where
$Ft(T1, zeta0, ur) = mketriple(tau1, zeta2, ur2) \wedge$
$Ft(T2, zeta2, ur2) = mketriple(tau2, zeta3, ur3) \wedge$
$issubtype(tau1) = TT \wedge$
$newtag(zeta3) = mkepair(nu, zeta4)$ by
$mketriple(mkearraytype(nu, tau1, tau2), zeta4, ur3)$;

$st5$: replace $At(mkerecord(ITlist), zeta0, zeta1)$ where
$distinct(selidlist(ITlist)) = TT \wedge$
$mkepair(taulist, zeta2) = Atstar(ITlist, zeta0, zeta1) \wedge$
$mkepair(nu, zeta3) = newtag(zeta2)$ by
$mkepair(mkerecordtype(nu, mkerecsec(selidlist(ITlist), taulist)), zeta3)$;

$Fst5$: replace $Ft(mkerecord(ITlist), zeta0, ur)$ where
$distinct(selidlist(ITlist)) = TT \wedge$

```
                mketriple(taulist, zeta2, ur1) = Ftstar(ITlist, zeta0, ur) ∧
                mkepair(nu, zeta3) = newtag(zeta2) by
              mketriple(
                 mkerecordtype(nu, mkerecsec(selidlist(ITlist), taulist)),
                 zeta3, ur1);

st6:  replace  At(mkepointer(I), zeta0, zeta1) where
                 newtag(zeta0) = mkepair(nu, zeta2) by
               mkepair(mkepointertype(nu, atype(apply1(zeta1, I))), zeta2);

Fst6: replace  Ft(mkepointer(I), zeta, ur) where
                 newtag(zeta) = mkepair(nu, zeta1) ∧
                 mketriple(zeta2, ur2, tau) = anonymous(zeta1, ur, I) by
               mketriple(mkepointertype(nu, tau), zeta2, ur2);

fstar1: replace Atstar(dl, z1, z2) where isnulllist(dl) = TT by z1;

fstar2: replace Atstar(dl, z1, z2) where
                 mkepair(tau, z3) = At(selfirst(dl), z1, z2)BY
```

1.2.2. Fixed points

```
rulefile (fixedpoints)

constant emptyset, emptyur;

%definition of resolve %

fix1:  replace resolve(a, m, t, e, z, ul)
       where ¬isnull(ul) ∧
         apply1(envrep(a, m, t, e, z), idof(ul)) =
           mketypemode(typerep(t, tau))
       by resolve(a, m, < t, ⊂ tpof(ul) ⊃, t ⊂ tau ⊃>, e, z, nextof(ul));

fix2:  replace resolve(a, m, t, e, z, ul) where isnull(ul) by t;

%resolve computes a least fixed point %

fix3:  replace fix(envFrep(a, m, t, e, z, ul)) by
         envrep(a, m, resolve(a, m, t, e, z, ul), e, z);
```


2. The program

The development of the program from specifications is straightforward. The refinement step for *Ce* is described in the text in some detail. Below we give a listing of the declarations for the static semantics implementation, the code of *Ce*, and routines used to compute recursive types.

2.1. Declarations

2.1.1. Types

```
pascal

type
      termnonterm =(tagindex, tagselect, taglabel, tagstmt,
                    tagcommandlist, tagtype, tagconst, tagvard,
                    tagconstdec, tagtypedec, tagvardec, tagfunction,
                    tagprocedure, tagvarp, tagvalp, tagenum,
                    tagtypeid, tagarray, tagrecord, tagsubrange,
                    tagpointer, tagprogram, tagif, tagassign,
                    tagwhile, tagrepeat, taggoto, tagpcall,
                    tagblock, tagdummy, tagezpid, tagnumber,
                    tagunop, tagbinop, tagderef, tagfcall);

    atree =↑ anode;
   anode = record
              skind:  termnonterm;
              subt1:  atree;
              subt2:  atree;
              subt3:  atree;
              subt4:  atree;
              next:   atree;
              info:   integer
            end ;

    ttype =↑ tnode;
    mode =↑ mnode;
   tnode = record
              tkind:  (tagrecordtype, tagarraytype,
                tagsubtype, tagpointertype,
                tagniltype);
              typetag:  integer;
```

```
                    lwb:  integer;
                    upb:  integer;
                    subt1:  ttype;
                    subt2:  ttype;
                    recs:  ttype;
                    id:  integer
                   end ;

          mnode = record
                    mkind:  (tagvalmode, tagvapmode, tagafuncmode,
                      tagsfuncmode, tagvarmode, tagtypemode,
                      tagprocmode, tagsprocmode, tagpfuncmode,
                      tagconstmode);
                    ty:  ttype;
                    mlist:  mode;
                    next:  mode;
                    val:  integer
                   end ;

static_environment = ↑enode;
           enode = record
                    id:  atree;
                    md:  mode;
                    next:  static_environment
                   end ;

  nullary_functions =(TT, FF, int, bool);

             uptr =↑unode;
            unode = record tp:  ttype; id:  atree; next:  uptr end ;
```

2.1.2. Abstract syntax

%Testing abstract syntax %

```
function Cisindex(e:atree):boolean;
global (#anode);
entry  true;
exit   truthrep(Cisindex) = isindex(synrep(#anode, e));
begin
  Cisindex ← e↑.skind = tagindex;
end ;
```



```
function Cisselect(e:atree):boolean;
global (#anode);
entry  true;
exit   truthrep(Cisselect) = isselect(synrep(#anode, e));
begin
  Cisselect ← e↑.skind = tagselect;
end ;

...

...
```

%Decomposition of abstract syntax %

```
...

procedure matchderef(e:atree; var e1:atree);
global (#anode);
entry  isderef(synrep(#anode, e)) = TT;
exit   mkederef(synrep(#anode, e1)) = synrep(#anode, e);
begin
  e1 ← e↑.subt1;
end ;

procedure matchfcall(e:atree; var i, elist:atree);
global (#anode);
entry  isfcall(synrep(#anode, e)) = TT;
exit   mkefcall(synrep(#anode, i), synrep(#anode, elist)) = synrep(#anode, e);
begin
  i ← e↑.subt1;
  elist ← e↑.subt2;
end ;
```

%testing types %

```
function Cisarraytype(tau:  ttype):  boolean;
global (#tnode);
entry  true;
exit   isarraytype(tunerep(#tnode, tau)) = truthrep(Cisarraytype);
begin
  Cisarraytype ← tau↑.tkind = tagarraytype;
end ;
```

```
...

%constructing new types %

procedure Cmkearraytype(nu: integer; tau1, tau2: ttype; var r: ttype);
global (#mnode, #tnode);
exit  typerep(#tnode, r) =
        mkearraytype(tagrep(nu), typerep(#tnode, tau1), typerep(#tnode, tau2));
begin
  new(r);
  r ↑ .typetag ← nu;
  r ↑ .tkind ← tagarraytype;
  r ↑ .subt1 ← tau1;
  r ↑ .subt2 ← tau2;
end ;

...

%decomposing types %

procedure matcharraytype(tau: ttype; var nu: integer; var tau1, tau2: ttype);
global (#tnode);
entry  isarraytype(typerep(#tnode, tau)) = TT;
exit  typerep(#tnode, tau) =
        mkearraytype(tagrep(nu), typerep(#tnode, tau1), typerep(#tnode, tau2));
begin
  nu ← tau ↑ .typetag;
  tau1 ← tau ↑ .subt1;
  tau2 ← tau ↑ .subt2;
end ;

...

%testing modes %

function Cisafuncmode(mu: mode): boolean;
global (#tnode, #mnode);
entry  true;
exit  isafuncmode(moderep(#mnode, #tnode, mu)) =
        truthrep(Cisafuncmode);
begin
```



```
  Cisafuncmode ← mu ↑ .mkind = tagafuncmode;
end ;

...

%constructing modes %

procedure Cmkeafuncmode(mul: mode; tau: ttype; var mu: mode);
global (#enode, #mnode, #tnode, #anode);
entry  true;
exit  moderep(#mnode, #tnode, mu) =
        mkeafuncmode(moderep(#mnode, #tnode, mul), typerep(#tnode, tau));
begin
  new(mu);
  mu ↑ .mkind ← tagafuncmode;
  mu ↑ .mlist ← mul;
  mu ↑ .ty ← tau;
end ;

...

%decomposing modes %

procedure matchafuncmode(mu: mode; var mulist: mode; var tau: ttype);
global (#mnode, #tnode);
entry  isafuncmode(moderep(#mnode, #tnode, mu)) = TT;
exit  moderep(#mnode, #tnode, mu) =
        mkeafuncmode(moderep(#mnode, #tnode, mulist),
            typerep(#tnode, tau));
begin
  tau ← mu ↑ .ty;
  mulist ← mu ↑ .mlist;
end ;

...
```

2.1.3. Auxiliary functions

```
procedure error;
entry true; exit false; external ;

function Co(op:atree; md:mode):mode;
global (#enode, #mnode, #tnode, #anode);
```

```
entry  true;
exit   moderep(#mnode, #tnode, Co) =
         Ao(synrep(#anode, op), moderep(#mnode, #tnode, md));
external ;

...

%routines for binary operators, special functions, special procedures%

...

function Cisint(Ty:ttype):boolean;
global (#enode, #mnode, #tnode, #anode);
entry  true;
exit   truthrep(Cisint) = isint(typerep(#tnode, Ty));
begin
Cisint ← (Ctypetag(ty) = int);
end ;

function Cisbool(Ty:ttype):boolean;

...

...

procedure Coverlaps(Ty1, Ty2:ttype; var r:boolean);
global (#enode, #mnode, #tnode, #anode);
exit   truthrep(r) = overlaps(typerep(#tnode, Ty1), typerep(#tnode, Ty2));
var
  nu1, nu2, iota1, iota2, iota3, iota4:integer;
begin
  if Cissubtype(Ty1) and Cissubtype(Ty2) then
  begin
    matchsubtype(Ty1, nu1, iota1, iota2);
    matchsubtype(Ty2, nu2, iota3, iota4);
     if nu1 = nu2 then r ← (iota1 ≤ iota4) and (iota3 ≤ iota2)
       else error;
  end  else error;
end ;

procedure Ccontains(Ty1, Ty2:ttype; var r:boolean);
```



```
...

function Cisreturnable(Ty:ttype):boolean;

...

procedure Cunion(Ty1, Ty2:ttype; var r:ttype);
global (#enode, #mnode, #tnode, #anode);
exit   typerep(#tnode, r) = Aunion(typerep(#tnode, Ty1), typerep(#tnode, Ty2));
var
  nu1, nu2, iota1, iota2, iota3, iota4:integer;
begin
  if Cissubtype(Ty1) and Cissubtype(Ty2) then
  begin
    matchsubtype(Ty1, nu1, iota1, iota2);
    matchsubtype(Ty2, nu2, iota3, iota4);
     if nu1 = nu2 then Cmkesubtype(nu1, iota1, iota4, r)
       else error
  end  else error;
end ;

...

...

procedure Cpassable(Md1, Md2:mode; var r:boolean);
global (#enode, #mnode, #tnode, #anode);
entry  true;
exit   truthrep(r)
         = passable(moderep(#mnode, #tnode, Md1),
           moderep(#mnode, #tnode, Md2));
var
  tau1, tau2:ttype;
  t:boolean;
begin
  Ctype(Md1, tau1); Ctype(md2, tau2);
   if Cisvar(Md1) then
   begin
     r ← Cisvar(Md2) and
       Cequal(tau1, tau2)
   end  else
   if Cisval(Md1) then
```

```
  begin
    Ccompatible(tau1, tau2, t);
    r ← Cisval(Md1) and t;
  end else begin error; r ← false; end ;
end ;
```

2.2. Expressions

```
procedure Cestar(ezp:atree; zeta:static_environment; var m:mode);
global (#enode, #mnode, #tnode, #anode);
entry  true;
exit   moderep(#mnode, #tnode, m) =
         Aestar(synrep(#anode, ezp),
           envrep(#anode, #mnode, #tnode, #enode, zeta));
forward ;

procedure Ce(ezp:atree; zeta:static_environment; var m:mode);
global (#enode, #mnode, #tnode, #anode);
entry  true;
exit   moderep(#mnode, #tnode, m) =
         Ae(synrep(#anode, ezp),
           envrep(#anode, #mnode, #tnode, #enode, zeta));
var
  b              :  boolean;
  I              :  atree;
  n              :  atree;
  nu             :  integer;
  iota           :  integer;
  md, m1, m2     :  mode;
  op             :  atree;
  e, e1, e2      :  atree;
  elist          :  atree;
  mulist2, mulist:  mode;
  psi, tauprime,
  tau, tau1, tau2 :  ttype;
begin
  if Cisnumber(ezp) then
  begin
    matchnumber(ezp, n);
    iota ← value(n);
    Cmkesubtype(int, iota, iota, tau);
    Cmkeconstmode(iota, tau, m);
  end else
```



```
  if Cisezpid(ezp) then
  begin
    matchezpid(ezp, I);
    m ← capply1(zeta, I);
    if Cistypemode(m) then error else
    if Cisprocmode(m) then error else
    if Cissprocmode(m) then error;
  end else
  if Cisunop(ezp) then
  begin
    matchunop(ezp, op, e);
    Ce(e, zeta, m);
    m ← Co(op, m);
  end else
  if Cisbinop(ezp) then
  begin
    matchbinop(ezp, e1, op, e2);
    Ce(e1, zeta, m1);
    Ce(e2, zeta, m2);
    m ← Cw(op, m1, m2);
  end else
  if Cisfcall(ezp) then
  begin
    matchfcall(ezp, i, elist);
    md ← capply1(zeta, i);
    if Cisafuncmode(md) then
    begin
      matchafuncmode(md, mulist, tau);
      Cestar(elist, zeta, mulist2);
      Cpassablestar(mulist, mulist2, b);
      if b then Cmkevalmode(tau, m) else error;
    end else
    if Cispfuncmode(md) then
    begin
      matchpfuncmode(md, mulist, tau);
      Cestar(elist, zeta, mulist2);
      Cpassablestar(mulist, mulist2, b);
      if b then Cmkevalmode(tau, m) else error;
    end else
    if Cisfuncmode(md) then
    begin
      Cestar(elist, zeta, mulist2);
```

```
      m ← Csf(i, mulist2);
    end else error;
  end else
  if Cisselect(ezp) then
  begin
    matchselect(ezp, e, i);
    Ce(e, zeta, md);
     if Cisvarmode(md) then
     begin
       matchvarmode(md, tau);
        if Cisrecordtype(tau) then
        begin
          matchrecordtype(tau, nu, psi);
          m ← assoc(psi, i);
        end else error;
     end else
     if Cisvapmode(md) then
     begin
       matchvapmode(md, tau);
        if Cisrecordtype(tau) then
        begin
          matchrecordtype(tau, nu, psi);
          m ← assoc(psi, i);
        end else error;
     end else error;
  end else
  if Cisindex(ezp) then
  begin
    matchindex(ezp, e1, e2);
    Ce(e1, zeta, m1);
    Ce(e2, zeta, m2);
    Ctype(m1, tau);
     if Cisarraytype(tau) then
     begin
       matcharraytype(tau, nu, tau1, tau2);
       Ctype(m2, tauprime);
       Ccompatible(tau1, tauprime, b);
        if b then
          if Cisvar(m1) then Cmkevarmode(tau2, m) else
          if Cisval(m1) then Cmkevalmode(tau2, m) else
            error
        else error
```



```
      end else error;
    end else
    if Cisderef(ezp) then
    begin
      matchderef(ezp, e);
      Ce(e, zeta, md);
      Ctype(Md, tau);
       if Cispointertype(tau) then
       begin
         matchpointertype(tau, nu, tauprime);
         Cmkevarmode(tauprime, m);
       end else error;
    end else error;
  end ;

  procedure Cestar;
  %has forward declaration %
  var mu1, mu2:mode;
  begin
    if Cisnulllist(ezp) then m ← nil else
    begin
      Ce(Cselfirst(ezp), zeta, mu1);
      Cestar(Cselrest(ezp), zeta, mu2);
      Cmodeappend(mu1, mu2, m);
    end ;
  end ;
```

2.3. Types

Some additional documentation not discussed in the text is included in the specifications of the following routine. This documentation is required to prove that changes to reference classes occur in an orderly fashion.

```
procedure CFt(tpe:atree; zeta:static_environment;
                var t:ttype; var z:static_environment;
                var ures:uptr);
global ( var #tnode, #unode; #enode, #anode, #mnode);
initial #tnode = tnode0, ures = ures0, #unode = unode0;
exit   mketriple(typercp(#tnode, t),
           envrcp(#anode, #mnode, #tnode, #enode, z),
           urcp(#unode, #anode, ures)) =
       Ft(synrcp(#anode, tpe),
         envrcp(#anode, #mnode, tnode0, #enode, zeta),
```

```
        urep(unode0, #anode, ures0)) ∧
      subclass(tnode0, #tnode) ∧
      ¬ptrto(setminus(pointers(urep(#unode, #anode, ures)),
            pointers(urep(unode0, #anode, ures0))), tnode0);
var
  n, nu:integer;
  e1, e2, id, idlist, idlist0:atree;
  mu, m1, m2:mode;
  zeta3, zeta4, zeta5:static_environment;
  rs:atree;
  recs, taulist:ttype;
  tau, t1, t2:ttype;
begin
  if Cistypeid(tpe) then
  begin
    matchtypeid(tpe, id);
    mu ← Capply1(zeta, id);
    if Cistypemode(mu) then
    begin
      matchtypemode(mu, tau);
      t ← tau;
      z ← zeta;
    end else error;
  end else
  if Cisenum(tpe) then
  begin
    matchenum(tpe, idlist);
    Cnewtag(zeta, nu, zeta3);
    n ← 0;
    idlist0 ← idlist;
    zeta4 ← zeta3;
    invariant etstar(synrep(#anode, idlist0), tagrep(nu),
                envrep(#anode, #mnode, #tnode, #enode, zeta3), intrep(0)) =
              etstar(synrep(#anode, idlist), tagrep(nu),
                envrep(#anode, #mnode, #tnode, #enode, zeta4), intrep(n))
    while not Cisnulllist(idlist) do
    begin
      n ← n + 1;
      Cmkesubtype(nu, n, n, tau);
      Cmkeconstmode(n, tau, mu);
      Credefine(zeta4, mu, Cselfirst(idlist), zeta4);
      idlist ← Cselrest(idlist);
```



```
    end ;
    Cmkesubtype(nu, 1, n, t);
    z ← zeta4;
  end else
  if Cissubrange(tpe) then
  begin
    matchsubrange(tpe, e1, e2);
    Ce(e1, zeta, m1);
    Ce(e2, zeta, m2);
    if Cisconstmode(m1) then
    if Cisconstmode(m2) then
    begin
      Ctype(m1, t1);
      Ctype(m2, t2);
      Cunion(t1, t2, t);
      z ← zeta;
    end else error else error;
  end else
  if Cisarray(tpe) then
  begin
    matcharray(tpe, e1, e2);
    CFt(e1, zeta, t1, zeta3, ures);
    CFt(e2, zeta3, t2, zeta4, ures);
    if Cissubtype(t1) then
    begin
      Cnewtag(zeta4, nu, zeta5);
      Cmkearraytype(nu, t1, t2, t);
      z ← zeta5;
    end else error;
  end else
  if Cisrecord(tpe) then
  begin
    matchrecord(tpe, rs);
    idlist ← Cselidlist(rs);
    if Cdistinct(idlist) then
    begin
      CFtstar(rs, zeta, taulist, zeta3, ures);
      Cnewtag(zeta3, nu, z);
      Cmkerecsec(idlist, taulist, recs);
      Cmkerecordtype(nu, recs, t);
    end else error
  end else
```

```
  if Cispointer(tpe) then
  begin
    matchpointer(tpe, id);
    Cnewtag(zeta, nu, z);
    addnewu(id, tau, ures, z);
    Cmkepointertype(nu, tau, t);
  end  else error;
end ;

procedure CFdt(decl:atree; zeta:static_environment;
                          var z:static_environment;
                          var ures:uptr);
global ( var #tnode, #unode; #enode, #anode, #mnode);
initial #tnode = tnode0, ures = ures0, #unode = unode0;
exit   mkepair(envrep(#anode, #mnode, #tnode, #enode, z),
           urep(#unode, #anode, ures)) =
       Fdt(synrep(#anode, decl),
         envrep(#anode, #mnode, tnode0, #enode, zeta),
           urep(unode0, #anode, ures0)) ∧
       subclass(tnode0, #tnode) ∧
         ¬ptrto(setminus(pointers(urep(#unode, #anode, ures)),
              pointers(urep(unode0, #anode, ures0))),
                tnode0);
var
  i, t:atree;
  tau:ttype;
  mu:mode;
begin
  if Cistypedec(decl) then
  begin
    matchtypedec(decl, i, t);
    CFt(t, zeta, tau, z, ures);
    Cmketypemode(tau, mu);
    Credefine(z, mu, i, z);
  end  else error;
end ;

procedure Cresolve(z:static_environment; ures:uptr);
global ( var #tnode; #mnode, #anode, #enode, #unode);
initial #tnode = tnode0;
exit   #tnode = resolve(#anode, #mnode, tnode0, #enode, z,
           urep(#unode, #anode, ures)) ∧
```



```
       uchanged(pointers(urep(#unode, #anode, ures)), tnode0, #tnode);
var
  r:uptr;
  mu:mode;
  tau:ttype;
begin
  r ← ures;
  invariant resolve(#anode, #mnode, tnode0, #enode, z,
              urep(#unode, #anode, ures)) =
            resolve(#anode, #mnode, #tnode, #enode, z,
              urep(#unode, #anode, r)) ∧
            uchanged(pointers(urep(#unode, #anode, ures)), tnode0, #tnode) ∧
            subset(pointers(urep(#unode, #anode, r)),
                pointers(urep(#unode, #anode, ures)))
  while r ≠ nil do
  begin
    mu ← Capply1(z, r ↑ .id);
      if Cistypemode(mu) then
      begin
        matchtypemode(mu, tau);
        r ↑ .tp ↑← tau ↑;
        r ← r ↑ .next;
      end  else error;
  end
end ;
```

Appendix 6. Code generation

1. Logical basis

rulefile (*code_generation*)

constant *nocode*, *nulllist*;

*T*1: infer *varthetaG*(*Cscrl*(*G*, *zeta*, *rho*, *gamma*), *s*,
Mscr(*Tmkeblock*(*z*), *rhoT*, *gammaT*))
from *ForallUGG*(*G*, *redeftrue*(*zeta*, *j*(*G*)), *yy*, *z*) ∧
varthetaU(*rho*, *zeta*, *y*, *rhoT*) ∧
s = *PhiS*(*zeta*, *rho*, *y*) ∧
disjointlabels(*yy*, *j*(*G*), *y*) ∧
VarthetaG(*gamma*, *s*, *gammaT*);

*T*2: infer *varthetaG*(*Bscr*(*mkeblock*(*Dlist*, *G*), *zeta*, *rho*, *gamma*), *s*,
Mscr(*Tmkeblock*(*z*), *rhoT*, *gammaT*))
from *ForallBLO*(*Dlist*, *G*, *d*(*Dlist*, *redeftrue*(*zeta*, *j*(*G*))), *yyy*, *z*) ∧
disjointlabels(*yy*, *j*(*G*), *y*) ∧
disjointlabels(*yyy*, *k*(*Dlist*), *yy*) ∧
s = *PhiS*(*zeta*, *rho*, *y*) ∧
varthetaU(*Vscrstar*(*Dlist*, *zeta*, *rho*), *zeta*, *y*, *rhoT*) ∧
VarthetaG(*gamma*, *s*, *gammaT*);

*T*3: infer *varthetaGstar*(*append*(*g*1, *g*2), *s*, *append*(*t*1, *t*2))
from *varthetaGstar*(*g*1, *s*, *t*1) ∧
varthetaGstar(*g*2, *s*, *t*2);

*T*4: infer *varthetaGstar*(*nulllist*, *s*, *nulllist*);

*T*5: infer *varthetaGstar*(*mkelist*(*z*), *s*, *mkelist*(*s*))
from *varthetaG*(*z*, *s*, *z*);

*appl*1: infer *varthetaG*(*Sapply*(*gamma*, *epsilon*), *s*, *Tapply*(*gammaT*, *epsilon*))
from *varthetaG*(*gamma*, *s*, *gammaT*);

*unop*1: infer *varthetaG*(*unop*(*gamma*, *O*), *s*, *Tunop*(*gammaT*, *O*))
from *varthetaG*(*gamma*, *s*, *gammaT*);

*binop*1: infer *varthetaG*(*binop*(*gamma*, *O*), *s*, *Tbinop*(*gammaT*, *O*))
from *varthetaG*(*gamma*, *s*, *gammaT*);

*addr*1: infer *varthetaG*(*saddr*(*gamma*, *alpha*, *n*), *s*,
tapply(*taddr*(*gammaT*, *m*), *alphaT*))
from *varthetaG*(*gamma*, *s*, *gammaT*) ∧ *n* = *m* ∧
smember(*n*, *alpha*, *alphaT*, *s*);

*vthU*1: infer *smember*(*apply2*(*rho*, *I*), *apply1*(*rho*, *I*),
yapply(*y*, *apply2*(*rho*, *I*), *I*), *PhiS*(*zeta*, *rho*, *y*))
from *varthetaU*(*rho*, *zeta*, *y*, *rhoT*);

rulefile (*target_semantics*)

constant *nocode*, *nulllist*;

*Mscr*1: replace *Mscr*(*nocode*, *rhoT*, *gammaT*) by *gammaT*;

*Lscr*1: replace *LscrT*(*nocode*, *rhoT*, *gammaT*) by *nulllist*;

*list*1: replace *cons*(*z*, *nulllist*) by *mkelist*(*z*);

rulefile (*commands*)

constant *nulllist*;

*com*1: replace *Cscr*(*mkelabel*(*N*, *Theta*), *zeta*, *rho*, *gamma*)
by *Bscr*(*Theta*, *zeta*, *rho*, *gamma*);

*com*2: replace *Cscr*(*mkestmt*(*Theta*), *zeta*, *rho*, *gamma*)
by *Bscr*(*Theta*, *zeta*, *rho*, *gamma*);

*com*3: replace *Cscr*(*mkecommandlist*(*G*0, *G*1), *zeta*, *rho*, *gamma*)
by *Cscr*(*G*0, *zeta*, *rho*, *Cscr*(*G*1, *zeta*, *rho*, *gamma*));

*Jscr*1: replace *Jscr*(*mkecommandlist*(*G*0, *G*1), *zeta*, *rho*, *gamma*)

```
            by append(Jscr(G0, zeta, rho, Cscr(G1, zeta, rho, gamma)),
               Jscr(G1, zeta, rho, gamma));

Jscr2: replace Jscr(mkestmt(S), zeta, rho, gamma) by nulllist;

Jscr3: replace Jscr(mkelabel(N, S), zeta, rho, gamma)
           by mkelist(Bscr(S, zeta, rho, gamma));

rulefile (statements)

constant mkedummy, mkesprocmode, mkenulllist;

Bd01: replace Bscr(mkeassign(E0, E1), zeta, rho, gamma)
        where ce(E0, zeta) = mkevarmode(tau)
          by Lscr(E0, zeta, rho, Ascr(E1, zeta, mkevalmode(tau), rho,
                update(gamma)));

Bd02: replace Bscr(mkeif(E, G0, G1), zeta, rho, gamma)
        where mkepair(zeta0, N0) = newlabel(zeta) ∧
              mkepair(zeta1, N1) = newlabel(zeta0)
          by Cscrl(mkecommandlist(mkestmt(mkeunless(E, N0)),
             mkecommandlist(G0,
             mkecommandlist(mkestmt(mkegoto(N1)),
             mkecommandlist(mkelabel(N0,
                 mkestmt(mkeblock(mkenulllist, G1))),
               mkestmt(mkedummy))))),
             zeta1, rho, gamma);

Bd03: replace Bscr(S, zeta, rho, gamma)
        where isdummy(S)
          by gamma;

Bd04: replace Bscr(mkewhile(E, G), zeta, rho, gamma)
        where mkepair(zeta0, N0) = newlabel(zeta) ∧
              mkepair(zeta1, N1) = newlabel(zeta0)
          by Cscrl(mkecommandlist(mkelabel(N0, mkeunless(E, N1)),
             mkecommandlist(G,
             mkecommandlist(mkestmt(mkegoto(N0)),
               mkelabel(N1, mkedummy)))),
             zeta1, rho, gamma);

Bd05: replace Bscr(mkerepeat(G, E), zeta, rho, gamma)
```



```
        where mkepair(zeta0, N0) = newlabel(zeta)
          by Cscrl(mkecommandlist(mkelabel(N0, mkeblock(mkenulllist, G)),
               mkestmt(mkeunless(E, N0))),
             zeta0, rho, gamma);

Bd06: replace Bscr(mkegoto(N), zeta, rho, gamma)
          by adjust(apply(rho, N), gamma);

Bd07: replace Bscr(mkepcall(I, Elist), zeta, rho, gamma)
        where issprocmode(apply(zeta, I))
          by Escrstar(Elist, zeta, rho, Sscrp(I, gamma));

Bd08: replace Bscr(mkepcall(I, Elist), zeta, rho, gamma)
        where ¬issprocmode(apply(zeta, I)) ∧ mkepair(mulist, n) =
                estar(Elist, zeta)
          by Ascrstar(Elist, zeta, mulist, rho, Spcall(rho, I, gamma));

Bd10: replace Bscr(mkeunless(E0, N0), zeta, rho, gamma)
          by Rscr(E0, zeta, rho, Cnd(gamma, apply(rho, N0)));

rulefile (expressions)

constant nulllist, mkesfunmode;

aux1: from issfunmode(x) infer x = mkesfunmode;

ed01: replace Escr(E, zeta, rho, gamma)
        where ce(E, zeta) = mkeconstmode(epsilon, tau)
          by sapply(gamma, epsilon);

ed02: replace Escr(E, zeta, rho, gamma)
        where ¬isconstmode(ce(E, zeta))
          by AEscr(E, zeta, rho, gamma);

ed03: replace AEscr(E, zeta, rho, gamma)
        where E = mkenumber(N) by sapply(gamma, N);

ed04: replace AEscr(E, zeta, rho, gamma)
        where E = mkeezpid(I) ∧
              apply(zeta, I) = mkesfunmode
          by Sscrf(I, gamma);
```

ed05: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkeezpid(I) \wedge$
$isafunmode(apply(zeta, I))$
by $fcall(apply3(rho, I), gamma)$;

ed06: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkeezpid(I) \wedge$
$ispfunmode(apply(zeta, I))$
by $fcall(apply3(rho, I), gamma)$;

ed07: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkeezpid(I) \wedge$
$\neg issfunmode(apply(zeta, I)) \wedge$
$\neg isafunmode(apply(zeta, I)) \wedge$
$\neg ispfunmode(apply(zeta, I))$
by $saddr(gamma, apply1(rho, I), apply2(rho, I))$;

ed08: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkeunop(O, E1)$
by $Rscr(E1, zeta, rho, unop(gamma, O))$;

ed09: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkebinop(E0, Omega, E1)$
by $Rscr(E0, zeta, rho, Rscr(E1, zeta, rho, binop(gamma, Omega)))$;

ed10: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkederef(E1)$
by $Rscr(E1, zeta, rho, verifyn(gamma))$;

ed11: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkefcall(I, Elist) \wedge$
$apply(zeta, I) = mkesfunmode$
by $Escrstar(Elist, zeta, rho, Sscrf(I, gamma))$;

ed12: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkefcall(I, Elist) \wedge$
$apply(zeta, I) = mkeafunmode(mulist, tau)$
by $Ascrstar(Elist, zeta, mulist, rho, fcall(apply3(rho, I), gamma))$;

ed13: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkefcall(I, Elist) \wedge$
$apply(zeta, I) = mkepfunmode(mulist, tau)$

by $Ascrstar(Elist, zeta, mulist, rho, fcall(apply3(rho, I), gamma))$;

ed14: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkeindex(E0, E1) \wedge$
$ctype(ce(E0, zeta)) = mkearraytype(nu, tau0, tau1)$
by $Escr(E0, zeta, rho, Ascr(E1, zeta, mkevalmode(tau0), rho,$
$index(gamma)))$;

ed15: replace $AEscr(E, zeta, rho, gamma)$
where $E = mkeselect(E1, I)$
by $Escr(E1, zeta, rho, select(I, gamma))$;

2. The program

2.1. Declarations

```
pascal
type Scontinuation      = integer;
     Senvironment       = integer;
     Storage_map        = integer;
     Tcontinuation      = integer;
     Tcontlist          = (nulllist);
     Tenvironment       = integer;
     asyn               = (mkenulllist, mkedummy);
     code               = (nocode);
     compile_environment= integer;
     function_value     = integer;
     location           = integer;
     mode               = integer;
     static_environment = integer;
     ttype              = integer,
     value              = integer;
```

2.2. Virtual procedures

Some of the valuations of the definition of *LS* and *LT* are required as virtual functions.

```
procedure error; entry true; exit false; external ;

function Ascr(E1: asyn; zeta: static_environment; mu: mode;
```

```
  rho: Senvironment; gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function Cscr(G: asyn; zeta: static_environment;
  rho: Senvironment; gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function Mscr(z: code; rhoT:
      Tenvironment; gammaT: Tcontinuation): Tcontinuation;
entry true; exit true; external ;

function Rscr(E0: asyn; zeta: static_environment;
      rho: Senvironment; gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function Spcall(rho: Senvironment; I: asyn;
      gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function Sscrp(I: asyn; gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function binop(gamma: Scontinuation; O: asyn): Scontinuation;
entry true; exit true; external ;

function ce(E: asyn; zeta: static_environment): mode;
entry true; exit true; external ;

function cnd(gamma0, gamma1: Scontinuation): Scontinuation;
entry true; exit true; external ;

function ctype(mu: mode): ttype;
entry true; exit true; external ;

function d(Dlist: asyn; zeta: static_environment): static_environment;
entry true; exit true; external ;

function fcall(pi: function_value; gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function index(gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;
```



```
function j(G: asyn): asyn;
entry true; exit true; external ;

function redef(y: compile_environment; n: asyn; m: integer):
    compile_environment;
entry true; exit true; external ;

function redeftrue(zeta: static_environment; j: asyn): static_environment;
entry true; exit true; external ;

function select(I: asyn; gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function unop(gamma: Scontinuation; O: asyn): Scontinuation;
entry true; exit true; external ;

function update(gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

function verifyn(gamma: Scontinuation): Scontinuation;
entry true; exit true; external ;

procedure cestar(Elist: asyn; zeta: static_environment;
     var mulist: mode; var m: integer);
exit mkepair(mulist, m) = estar(Elist, zeta);
external ;
```

2.3. Auxiliary functions

```
function apply(zeta: static_environment; I: asyn): mode;
entry true; exit true; external ;

function yapply(y: compile_environment; n: integer; I: asyn): location;
entry true; exit true; external ;

function apply2(rho: Senvironment; I: asyn): integer;
entry true; exit true; external ;

function apply3(rho: Senvironment; I: asyn): function_value;
entry true; exit true; external ;
```

```
procedure mkeTappend(z2: code; z1: code; var z: code;
    rhoT: Tenvironment; gammaT: Tcontinuation);
exit Mscr(z, rhoT, gammaT) = Mscr(z2, rhoT, Mscr(z1, rhoT, gammaT)) ∧
     LscrT(z, rhoT, gammaT) =
       append(LscrT(z2, rhoT, Mscr(z1, rhoT, gammaT)),
         LscrT(z1, rhoT, gammaT));
external ;

procedure mkeTlabels(y: compile_environment; l: asyn;
       var ynew: compile_environment);
entry true;
exit disjointlabels(ynew, l, y);
external ;

procedure newsourcelabel( var N0: asyn; zeta: static_environment;
     var zeta0: static_environment);
entry true;
exit mkepair(zeta0, N0) = newlabel(zeta);
external ;
```

2.4. Abstract syntax, types and modes

Functions to test, access, and construct objects of abstract syntax, modes and types are identical to those used in the static semantics. No refinement is given here.

```
function isafunmode(mu: mode): boolean;
entry true; exit true; external ;

function isarraytype(tau: ttype): boolean;
entry true; exit true; external ;

function isassign(Stmt: asyn): boolean;
entry true; exit true; external ;

function isbinop(E: asyn): boolean;
entry true; exit true; external ;

...

...

function isvarmode(mu: mode): boolean;
```



```
entry true; exit true; external ;

function iswhile(Stmt: asyn): boolean;
entry true; exit true; external ;

function mkeblock(D, G: asyn): asyn;
exit true; external ;

function mkecommandlist(G0, G1: asyn): asyn;
exit true; external ;

...

...

function mkevalmode(tau: ttype): mode;
entry true; exit true; external ;

procedure matchafunmode(mu: mode; var mulist: mode; var tau: ttype);
entry true;
exit mu = mkeafunmode(mulist, tau);
external ;

procedure matcharraytype(tau: ttype; var nu: integer;
                         var tau0: ttype; var tau1: ttype);
entry true;
exit tau = mkearraytype(nu, tau0, tau1);
external ;

...

...

procedure matchvarmode(mu: mode; var tau: ttype);
exit mkevarmode(tau) = mu;
external ;

procedure matchwhile(Stmt: asyn; var E0: asyn; var G: asyn);
exit mkewhile(E0, G) = Stmt;
external ;
```

2.5. Code generating functions

We assume the following code generation procedures to be external. An implementation is immediate in most cases.

```
procedure Acode(E: asyn;
                zeta: static_environment;
                mu: mode;
                rho: Senvironment;
                gamma: Scontinuation;
                y: compile_environment;
                rhoT: Tenvironment;
                gammaT: Tcontinuation;
                 var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Ascr(E, zeta, mu, rho, gamma),
      PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
external ;

procedure Acodestar(Elist: asyn;
                    zeta: static_environment;
                    mulist: mode;
                    rho: Senvironment;
                    gamma: Scontinuation;
                    y: compile_environment;
                    rhoT: Tenvironment;
                    gammaT: Tcontinuation;
                     var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Ascrstar(Elist, zeta, mulist, rho, gamma),
      PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
external ;

procedure Allocate(Dlist: asyn;
                   zeta: static_environment;
                   rhoT: Tenvironment;
                    var rho: Senvironment;
                   yold: compile_environment;
                    var y: compile_environment);
initial  rho = rho0;
   entry varthetaU(rho, zeta, yold, rhoT);
    exit varthetaU(rho, y, rhoT) ∧ rho = Vscrstar(Dlist, zeta, rho0);
external ;

procedure Bcode(Theta: asyn;
                zeta: static_environment;
                rho: Senvironment;
                gamma: Scontinuation;
                y: compile_environment;
                rhoT: Tenvironment;
                gammaT: Tcontinuation;
                 var z: code);
initial  z = z0;
  entry varthetaU(rho, zeta, y, rhoT) ∧
        varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
   exit varthetaG(Bscr(Theta, zeta, rho, gamma), PhiS(zeta, rho, y),
            Mscr(z, rhoT, gammaT)) ∧
        LscrT(z, rhoT, gammaT) = LscrT(z0, rhoT, gammaT);
external ;

procedure Clcode(T: asyn; zeta: static_environment; rho: Senvironment;
                 gamma: Scontinuation; y: compile_environment;
                 rhoT: Tenvironment; gammaT: Tcontinuation; var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Cscrl(T, zeta, rho, gamma), PhiS(zeta, rho, y),
          Mscr(z, rhoT, gammaT));
external ;

procedure Condcode(N: asyn; zeta: static_environment; rho: Senvironment;
                   gamma: Scontinuation; y: compile_environment;
                   rhoT: Tenvironment; gammaT: Tcontinuation; var z: code);
entry varthetaU(rho, zeta, s, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Cnd(gamma, apply(rho, N)), PhiS(zeta, rho, y),
          Mscr(z, rhoT, gammaT));
external ;

procedure Dstarcode(Dlist: asyn; zeta: static_environment;
                    rho: Senvironment,
                    gamma: Scontinuation, y: compile_environment,
                    rhoT: Tenvironment; gammaT: Tcontinuation; var z: code);
```

```
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Dscrstar(Dlist, zeta, rho, gamma), PhiS(zeta, rho, y),
          Mscr(z, rhoT, gammaT));
external ;

procedure Estarcode(Elist: asyn;
                    zeta: static_environment;
                    rho: Senvironment;
                    gamma: Scontinuation;
                    y: compile_environment;
                    rhoT: Tenvironment;
                    gammaT: Tcontinuation;
                     var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Escrstar(Elist, zeta, rho, gamma),
      PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
external ;

procedure Fstarcode(Dlist: asyn; zeta: static_environment;
                    rho: Senvironment;
                    y: compile_environment;
                    rhoT: Tenvironment; gammaT: Tcontinuation; var z: code);
entry varthetaU(rho, zeta, y, rhoT);
exit  varthetaU(Fscrstar(Dlist, zeta, rho), PhiS(zeta, rho, y), rhoT);
external ;

procedure Lcode(E0: asyn; zeta: static_environment;
                rho: Senvironment;
                gamma: Scontinuation; y: compile_environment;
                rhoT: Tenvironment; gammaT: Tcontinuation; var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Lscr(E0, zeta, rho, gamma), PhiS(zeta, rho, y),
          Mscr(z, rhoT, gammaT));
external ;

procedure Rcode(E: asyn;
                zeta: static_environment;
                rho: Senvironment;
                gamma: Scontinuation;
```



```
                y: compile_environment;
                rhoT: Tenvironment;
                gammaT: Tcontinuation;
                 var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Rscr(E, zeta, rho, gamma),
      PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
external ;

procedure Sfcode(I: asyn;
                 zeta: static_environment;
                 rho: Senvironment;
                 gamma: Scontinuation;
                 y: compile_environment;
                 rhoT: Tenvironment;
                 gammaT: Tcontinuation;
                  var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Sscrf(I, gamma), PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
external ;

procedure Spcallcode(rho: Senvironment; I: asyn;
                     gamma: Scontinuation; s: storage_map;
                     rhoT: Tenvironment; gammaT: Tcontinuation;
                      var z: code);
entry varthetaG(gamma, s, Mscr(z, rhoT, gammaT));
exit  varthetaG(Spcall(rho, I, gamma), s, Mscr(z, rhoT, gammaT));
external ;

procedure Sscrpcode(I: asyn; gamma: Scontinuation; s: storage_map;
                    rhoT: Tenvironment; gammaT: Tcontinuation; var z: code);
entry varthetaG(gamma, s, Mscr(z, rhoT, gammaT));
exit  varthetaG(Sscrp(I, gamma), s, Mscr(z, rhoT, gammaT));
external ;

procedure blockcode(Dlist: asyn;
                    G: asyn;
                    zeta: static_environment;
                    y: compile_environment;
                     var z: code);
```

```
entry true;
exit  forallBLO(Dlist, G, zeta, y, z);
external ;
```

The following functions generate primitive code sequences; i.e. single instructions or instruction sequences to access variables, index arrays and so on.

```
procedure cmkeaddr(N: value; rhoT: Tenvironment; gammaT: Tcontinuation;
                   var z: code);
initial  z = z0;
  entry true;
   exit Mscr(z, rhoT, gammaT) = Taddr(Mscr(z0, rhoT, gammaT), n);
external ;

procedure cmkebinop(O: asyn; rhoT: Tenvironment; gammaT: Tcontinuation;
                    var z: code);
initial  z = z0;
  entry true;
   exit Mscr(z, rhoT, gammaT) = Tbinop(Mscr(z0, rhoT, gammaT), O);
external ;

procedure cmkeblockcode(znew: code; var z: code;
                        rhoT: Tenvironment;
                        gammaT: Tcontinuation);
initial  z = z0;
  entry true;
   exit Mscr(z, rhoT gammaT) =
        Mscr(Tmkeblock(znew), rhoT, Mscr(z0, rhoT, gammaT));
external ;

procedure cmkelabelcode(n: integer;
                        rhoT: Tenvironment;
                        gammaT: Tcontinuation;
                        var z: code);
initial  z = z0;
  entry true;
   exit Mscr(z0, rhoT, gammaT) = Mscr(z, rhoT, gammaT) ∧
        LscrT(z, rhoT, gammaT) =
          cons(Mscr(z0, rhoT, gammaT), LscrT(z0, rhoT, gammaT));
external ;
```



```
procedure cmkelit(epsilon: value; rhoT: Tenvironment; gammaT: Tcontinuation;
                  var z: code);
initial  z = z0;
  entry true;
   exit Mscr(z, rhoT, gammaT) = Tapply(Mscr(z0, rhoT, gammaT), epsilon);
external ;

procedure cmkeunop(O: asyn; rhoT: Tenvironment; gammaT: Tcontinuation;
                   var z: code);
initial  z = z0;
  entry true;
   exit Mscr(z, rhoT, gammaT) = Tunop(Mscr(z0, rhoT, gammaT), O);
external ;

procedure fcallcode(I: asyn;
                    zeta: static_environment;
                    rho: Senvironment;
                    gamma: Scontinuation;
                    y: compile_environment;
                    rhoT: Tenvironment;
                    gammaT: Tcontinuation;
                    var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(fcall(apply3(rho, I), gamma), PhiS(zeta, rho, y),
        Mscr(z, rhoT, gammaT));
external ;

procedure indexcode(zeta: static_environment;
                    rho: Senvironment;
                    gamma: Scontinuation;
                    y: compile_environment;
                    rhoT: Tenvironment;
                    gammaT: Tcontinuation;
                    var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(index(gamma), PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
external ;

procedure jumpcode(N: asyn; rho: Senvironment;
                   gamma: Scontinuation; s: storage_map;
```

```
                    rhoT: Tenvironment; gammaT: Tcontinuation;
                     var z: code);
entry varthetaG(gamma, s, Mscr(z, rhoT, gammaT));
exit  varthetaG(adjust(apply(rho, N), gamma), s, Mscr(z, rhoT, gammaT));
external ;

procedure selectcode(I: asyn;
                    zeta: static_environment;
                    rho: Senvironment;
                    gamma: Scontinuation;
                    y: compile_environment;
                    rhoT: Tenvironment;
                    gammaT: Tcontinuation;
                     var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(select(I, gamma), PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
external ;

procedure updatecode(gamma: Scontinuation; s: storage_map;
                    rhoT: Tenvironment; gammaT: Tcontinuation;
                     var z: code);
entry varthetaG(gamma, s, Mscr(z, rhoT, gammaT));
exit  varthetaG(update(gamma), s, Mscr(z, rhoT, gammaT));
external ;

procedure verifyncode(zeta: static_environment;
                     rho: Senvironment;
                     gamma: Scontinuation;
                     y: compile_environment;
                     rhoT: Tenvironment;
                     gammaT: Tcontinuation;
                      var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(verifyn(gamma), y, Mscr(z, rhoT, gammaT));
external ;
```

2.6. Expressions

```
procedure AEcode(E: asyn;
```



```
                    zeta: static_environment;
                    rho: Tenvironment;
                    gamma: Scontinuation;
                    y: compile_environment;
                    rhoT: Tenvironment;
                    gammaT: Tcontinuation;
                     var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(AEscr(E, zeta, rho, gamma), PhiS(zeta, rho, y),
          Mscr(z, rhoT, gammaT));
forward ;

procedure Ecode(E: asyn;
                    zeta: static_environment;
                    rho: Tenvironment;
                    gamma: Scontinuation;
                    y: compile_environment;
                    rhoT: Tenvironment;
                    gammaT: Tcontinuation;
                     var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Escr(E, zeta, rho, gamma), PhiS(zeta, rho, y),
          Mscr(z, rhoT, gammaT));
var epsilon: value;
    mu: mode;
    tau: ttype;
begin
  mu ← ce(E, zeta);
   if isconstmode(mu) then
   begin
     matchconstmode(mu, epsilon, tau);
     cmkelit(epsilon, rhoT, gammaT, z)
   end  else
  AEcode(E, zeta, rho, gamma, y, rhoT, gammaT, z);
end ;


procedure AEcode;
%hasbeenforwarded%
var E1, E2, Elist, I, N, O: asyn;
```

```
    alpha: location;
    epsilon: value;
    mu, mulist: mode;
    nu, m: integer;
    tau, tau0, tau1: ttype;
begin
  if isnumber(E) then
  begin
    matchnumber(E, N);
    cmkelit(N, rhoT, gammaT, z);
  end else
  if isezpid(E) then
  begin
    matchezpid(E, I);
    mu ← apply(zeta, I);
    if issfunmode(mu) then
      Sfcode(I, zeta, rho, gamma, y, rhoT, gammaT, z)
    else
    if isafunmode(mu) then
    begin
      fcallcode(I, zeta, rho, gamma, y, rhoT, gammaT, z)
    end else
    if ispfunmode(mu) then
    begin
      fcallcode(I, zeta, rho, gamma, y, rhoT, gammaT, z)
    end else
    begin
      m ← apply2(rho, I);
      alpha ← yapply(y, m, I);
      cmkeaddr(m, rhoT, gammaT, z);
      cmkelit(alpha, rhoT, gammaT, z);
    end
  end else
  if isunop(E) then
  begin
    matchunop(E, O, E1);
    cmkeunop(O, rhoT, gammaT, z);
    Rcode(E1, zeta, rho, unop(gamma, O), y, rhoT, gammaT, z);
  end else
  if isbinop(E) then
  begin
    matchbinop(E, E1, O, E2);
```



```
    cmkebinop(O, rhoT, gammaT, z);
    Rcode(E2, zeta, rho, binop(gamma, O), y, rhoT, gammaT, z);
    Rcode(E1, zeta, rho, Rscr(E2, zeta, rho, binop(gamma, O)),
      y, rhoT, gammaT, z);
  end else
  if isderef(E) then
  begin
    matchderef(E, E1);
    verifyncode(zeta, rho, gamma, y, rhoT, gammaT, z);
    Rcode(E1, zeta, rho, verifyn(gamma), y, rhoT, gammaT, z);
  end else
  if isfcall(E) then
  begin
    matchfcall(E, I, Elist);
    mu ← apply(zeta, I);
    if issfunmode(mu) then
    begin
      Sfcode(I, zeta, rho, gamma, y, rhoT, gammaT, z);
      Ecodestar(Elist, zeta, rho,
          Sscrf(I, gamma), y, rhoT, gammaT, z);
    end else
    if isafunmode(mu) then
    begin
      matchafunmode(mu, mulist, tau);
      fcallcode(I, zeta, rho, gamma, y, rhoT, gammaT, z);
      Acodestar(Elist, zeta, mulist, rho,
          fcall(apply3(rho, I), gamma),
          y, rhoT, gammaT, z);
    end else
    if ispfunmode(mu) then
    begin
      matchafunmode(mu, mulist, tau);
      fcallcode(I, zeta, rho, gamma, y, rhoT, gammaT, z);
      Acodestar(Elist, zeta, mulist, rho,
          fcall(apply3(rho, I), gamma),
          y, rhoT, gammaT, z);
    end else error
  end else
  if isindex(E) then
  begin
    matchindex(E, E1, E2);
    mu ← ce(E1, zeta);
```

```
    tau ← ctype(mu);
    if isarraytype(tau) then
    begin
      matcharraytype(tau, nu, tau0, tau1);
      indexcode(zeta, rho, gamma, y, rhoT, gammaT, z);
      Acode(E2, zeta, mkevalmode(tau0), rho,
        index(gamma), y, rhoT, gammaT, z);
      Ecode(E1, zeta, rho,
        Ascr(E2, zeta, mkevalmode(tau0), rho,
          index(gamma)),
        y, rhoT, gammaT, z);
    end else error
  end else
  if isselect(e) then
  begin
    matchselect(E, E1, I);
    selectcode(I, zeta, rho, gamma, y, rhoT, gammaT, z);
    Ecode(E1, zeta, rho, select(I, gamma), y, rhoT, gammaT, z);
  end else error;
end ;
```

2.7. Commands

```
procedure Clcode(G: asyn;
                 zeta: static_environment;
                 rho: Senvironment;
                 gamma: Scontinuation;
                 y: compile_environment;
                 rhoT: Tenvironment;
                 gammaT: Tcontinuation;
                  var z: code);
initial z = z0;
entry  varthetaU(rho, zeta, y, rhoT) ∧
       varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit   varthetaG(Cscrl(G, zeta, rho, gamma), PhiS(zeta, rho, y),
       Mscr(z, rhoT, gammaT));
var znew: code;
    zetap: static_environment;
    ynew: compile_environment;
begin
  zetap ← redeftrue(zeta, j(G));
```



```
  mkeTlabels(y, j(G), ynew);
  Ccode(G, zetap, ynew, znew);
  cmkeblockcode(znew, z, rhoT, gammaT);
end ;


procedure Ccode(G: asyn;
                zeta: static_environment;
                rho: Senvironment;
                gamma: Scontinuation;
                y: compile_environment;
                rhoT: Tenvironment;
                gammaT: Tcontinuation;
                 var z: code);
initial z = z0;
entry  varthetaU(rho, zeta, y, rhoT) ∧
       varthetaG(gamma, PhiS(zeta, rho, y), gammaT);
exit   varthetaG(Cscr(G, zeta, rho, gamma), PhiS(zeta, rho, y),
           Mscr(z, rhoT, gammaT)) ∧
       varthetaGstar(Jscr(G, zeta, rho, gamma), PhiS(zeta, rho, y),
           LscrT(z, rhoT, gammaT));
var G0, G1, Theta, N: asyn;
    m: integer;
    z1, z2: code;
begin
  if islabel(G) then
  begin
    matchlabel(G, N, Theta);
    z ← nocode;
    Bcode(Theta, zeta, rho, gamma, y, rhoT, gammaT, z);
    m ← apply(y, n);
    cmkelabelcode(m, rhoT, gammaT, z);
  end else
  if isstmt(G) then
  begin
    matchstmt(G, Theta);
    z ← nocode;
    Bcode(Theta, zeta, rho, gamma, y, rhoT, gammaT, z);
  end else
  if iscommandlist(G) then
  begin
    matchcommandlist(G, G0, G1);
```

```
    Ccode(G1, zeta, rho, gamma, y, rhoT, gammaT, z1);
    Ccode(G0, zeta, rho, Cscr(G1, zeta, rho, gamma), y, rhoT,
      Mscr(z1, rhoT, gammaT), z2);
    mkeTappend(z2, z1, z, rhoT, gammaT);
  end else error
end ;


procedure Ccode(G: asyn;
                zeta: static_environment;
                y: compile_environment;
                 var z: code);
initial z = z0;
  exit forallUGG(G, zeta, y, z);
external ;
```

2.8. Statements

```
procedure Bcode(Stmt: asyn;
                zeta: static_environment;
                rho: Senvironment;
                gamma: Scontinuation;
                y: compile_environment;
                rhoT: Tenvironment;
                gammaT: Tcontinuation;
                 var z: code);
entry varthetaU(rho, zeta, y, rhoT) ∧
      varthetaG(gamma, PhiS(zeta, rho, y), Mscr(z, rhoT, gammaT));
exit  varthetaG(Bscr(Stmt, zeta, rho, gamma), PhiS(zeta, rho, y),
          Mscr(z, rhoT, gammaT));
var Dlist, E0, E1, Elist, G, G0, G1, I, N, N0, N1, T: asyn;
    m: integer;
    mu, mulist: mode;
    tau: ttype;
    zeta0, zeta1: static_environment;
    znew: code;
    ynew: compile_environment;
begin
  if isassign(Stmt) then
  begin
    matchassign(Stmt, E0, E1);
    mu ← ce(E0, zeta);
    if isvarmode(mu) then
    begin
      matchvarmode(mu, tau);
      updatecode(gamma, PhiS(zeta, rho, y), rhoT, gammaT, z);
      Acode(E1, zeta, mkevalmode(tau), rho, update(gamma),
        y, rhoT, gammaT, z);
      Lcode(E0, zeta, rho, Ascr(E1, zeta, mkevalmode(tau), rho,
            update(gamma)),
        y, rhoT, gammaT, z);
    end else error
  end else
  if isunless(Stmt) then
  begin
    matchunless(Stmt, E0, N);
    Condcode(N, zeta, rho, gamma, y, rhoT, gammaT, z);
    Rcode(E0, zeta, rho, Cnd(gamma, apply(rho, N)), y, rhoT, gammaT, z);
  end else
  if isif(Stmt) then
  begin
    matchif(Stmt, E0, G0, G1);
    newsourcelabel(N0, zeta, zeta0);
    newsourcelabel(N1, zeta0, zeta1);
    T ← mkecommandlist(mkestmt(mkeunless(E0, N0)),
    mkecommandlist(G0,
    mkecommandlist(mkestmt(mkegoto(N1)),
    mkecommandlist(mkelabel(N0, mkestmt(mkeblock(mkenullist, G1))),
      mkestmt(mkedummy)))));
    Clcode(T, zeta1, rho, gamma, y, rhoT, gammaT, z);
  end else
  if isdummy(Stmt) then begin end else
  if iswhile(Stmt) then
  begin
    matchwhile(Stmt, E0, G);
    newsourcelabel(N0, zeta, zeta0);
    newsourcelabel(N1, zeta0, zeta1);
    T ← mkecommandlist(mkelabel(N0, mkeunless(E0, N1)),
    mkecommandlist(G,
    mkecommandlist(mkestmt(mkegoto(N0)),
      mkelabel(N1, mkedummy))));
    Clcode(T, zeta1, rho, gamma, y, rhoT, gammaT, z);
  end else
```

```
  if isrepeat(Stmt) then
  begin
    matchrepeat(Stmt, G, E0);
    newsourcelabel(N0, zeta, zeta0);
    T ← mkecommandlist(mkelabel(N0, mkeblock(mkenullist, G)),
      mkestmt(mkeunless(E0, N0)));
    Clcode(T, zeta0, rho, gamma, y, rhoT, gammaT, z);
  end else
  if isgoto(Stmt) then
  begin
    matchgoto(Stmt, N);
    jumpcode(N, rho, gamma, PhiS(zeta, rho, y), rhoT, gammaT, z);
  end else
  if ispcall(Stmt) then
  begin
    matchpcall(Stmt, I, Elist);
    mu ← apply(zeta, I);
    if issprocmode(mu) then
    begin
      Sscrpcode(I, gamma, PhiS(zeta, rho, y), rhoT, gammaT, z);
      Estarcode(Elist, zeta, rho, Sscrp(I, gamma),
        y, rhoT, gammaT, z);
    end else
    begin
      cestar(Elist, zeta, mulist, m);
      Spcallcode(rho, I, gamma, PhiS(zeta, rho, y), rhoT, gammaT, z);
      Acodestar(Elist, zeta, mulist, rho, Spcall(rho, I, gamma),
        y, rhoT, gammaT, z);
    end
  end else
  if isblock(Stmt) then
  begin
    matchblock(Stmt, Dlist, G);
    Allocate(Dlist, zeta, rhoT, rho, y, ynew);
    zeta0 ← d(Dlist, redeftrue(zeta, j(G)));
    blockcode(Dlist, G, zeta0, ynew, znew);
    cmkeblockcode(znew, z, rhoT, gammaT);
  end else error;
end ;
```